Programok Statikus és Dinamikus Analízise

Ph.D. értekezés

Gergely Tamás

Témavezető
Dr. Gyimóthy Tibor

Szegedi Tudományegyetem
Informatika Doktori Iskola

Szoftverfejlesztés Tanszék
Természettudományi és Informatikai Kar
Szegedi Tudományegyetem

2010
Szeged

Előszó

Programszeletelés és hatásanalízis, disszertációm két témája. Két téma amely tulajdonképpen mégiscsak egy. Mindkettő a programelemek közötti kapcsolatok felderítését tűzte ki célul. Az egyik gyakorlat-orientált, ezért kisebb számításigényű, de kevésbé precíz, kevésbé részletes eredményt ad. A másik pontos, részletes eredményt szolgáltat, ezért számításigényes, így viszont a gyakorlatban kevésbé használt. A szoftverfejlesztési életciklus lépéseinek ugyanazon tevékenységeihez használhatóak fel, legfeljebb bizonyos feladatokra az egyik sokkal alkalmasabb mint a másik.

Az értekezés olyan eredményeket foglal össze, melyek teljes megvalósítása csapatmunka eredménye volt, amelyben néhol ugyan világos határok húzódtak az egyes résztvevők munkái között, de ezek mégis csak együtt nyertek értelmet. Köszönet témavezetőmnek Gyimóthy Tibornak az útmutatásért, szerzőtársaimnak Beszédes Árpádnak, Faragó Csabának, Tóth Gabriellának, Jász Juditnak, Fischer Ferencnek, Szabó Zsoltnak, Szegedi Attilának, Faragó Szabolcsnak, Václav Rajlichnak, Csirik Jánosnak, és munkatársaimnak Havasi Ferencnek, Kiss Ákosnak, Vidács Lászlónak, Siket Istvánnak, Ferenc Rudolfnak és a többieknek a közös munkáért.

Külön köszönet jár családomnak, feleségemnek, gyermekeimnek és szüleimnek türelmükért és támogatásukért.

Tartalomjegyzék

1. Bevezetés **1**
1.1. Eredmények összegzése 2
1.1.1. Magas szintű analízis 3
1.1.2. Alacsony szintű analízis 4
1.2. A dolgozat felépítése 5

I. Magas szintű analízis **7**

2. Hatásanalízis **9**
2.1. Hatásanalízis 9

3. A Static Execute After/Before (*SEA/SEB*) relációk **13**
3.1. Motiváció 13
3.1.1. A SEA lehetséges alkalmazásai 14
3.2. A *SEA/SEB* relációk 16
3.3. Az ICCFG gráf 17
3.4. A SEA kiszámítása 18
3.4.1. A SEA javításai 20
3.5. Összefoglalás 20

4. A Dynamic Function Coupling (*DFC*) metrika **21**
4.1. Motiváció 21
4.2. A *DFC* metrika 22
4.3. Algoritmusok 26
4.3.1. Globális algoritmus 26
4.3.2. Igényvezérelt algoritmus 26
4.3.3. Igényvezérelt algoritmus *d=1*-re 30
4.4. Mérések és kiértékelés 33

4.4.1. Mérési környezet 33
4.4.2. Pontosság és teljesség 34
4.4.3. Hatáshalmazok méretei 36
4.4.4. Kiértékelés 36
4.5. Összefoglalás 37

II. Alacsony szintű analízis 39

5. Programszeletelés 41
5.1. Szeletelés fogalma 42
5.2. Dinamikus szeletelés 44
5.3. Uniós szeletek 45

6. *d:U* alapú programszeletelési algoritmusok 47
6.1. Az algoritmusok osztályozása 48
6.2. Algoritmusok 49
6.2.1. Igényvezérelt algoritmusok 51
6.2.2. Praktikus algoritmusok 55
6.2.3. Párhuzamos algoritmusok 60
6.3. DDG ekvivalencia 64
6.3.1. Igényvezérelt algoritmusok 68
6.3.2. Praktikus algoritmusok 69
6.3.3. Párhuzamos algoritmusok 70
6.4. Összefoglalás 71

7. Szeletelő algoritmusok implementációja 73
7.1. Valódi programozási nyelvek szeletelésének problémái 74
7.1.1. Változók, mutatók és egyéb összetett adatstruktúrák 74
7.1.2. Vezérlési függőségek 75
7.1.3. Eljárások közötti analízis 76
7.1.4. Többszálúság 76
7.1.5. Kivételkezelés 76
7.1.6. Nyomkövetési információ 77
7.2. Megoldások C nyelvre 77
7.2.1. *d:U* átalakítása 77
7.2.2. Változók, mutatók 78

7.2.3. Módosított algoritmus C nyelvre 84
7.3. Java implementáció 89
7.3.1. A keretrendszer 89
7.3.2. Statikus elemző 91
7.3.3. Dinamikus szeletek számítása 91
7.3.4. Uniós szeletek és lefedettség számítása 92
7.4. Összefoglalás 92

8. *d:U* alapú szeletelő algoritmusok kiértékelése 93
8.1. Szeletelő algoritmusok komplexitásai 93
8.1.1. A DDG alapú algoritmus komplexitása 95
8.1.2. Igényvezérelt algoritmusok 95
8.1.3. Praktikus algoritmusok 96
8.1.4. Párhuzamos algoritmusok 97
8.1.5. Komplexitások összehasonlítása 98
8.2. Mérési eredmények 100
8.2.1. Mérések C nyelvű implementációval 100
8.2.2. Mérések Java nyelvű implementációval 104
8.3. Összefoglalás 110

9. Eredmények összegzése 111

Függelékek 113

A. Magyar nyelvű összefoglaló 115

B. Summary in English 119

Ábrák jegyzéke

3.1. Példa: rejtett függőség 15
3.2. Példa: ICCFG 18
3.3. ComputeSEA algoritmus 19

4.1. Példa: dinamikus hívási fa 25
4.2. ComputeDFC algoritmus: főprogram 27
4.3. ComputeDFC algoritmus: távolságszámítás 27
4.4. ComputeER algoritmus: főprogram 28
4.5. ComputeER algoritmus: belépési esemény 29
4.6. ComputeER algoritmus: visszatérési esemény 30
4.7. ComputeFirewallER algoritmus: főprogram 31
4.8. ComputeFirewallER algoritmus: belépési esemény 32
4.9. ComputeFirewallER algoritmus: visszatérési esemény 32
4.10. Az ER reláció pontossága és teljessége 35

5.1. Realizálható szeletek közelítése 45

6.1. Példa: egy példaprogram és $d : U$ reprezentációja 52
6.2. Példa: a példaprogram végrehajtási nyoma az $a = 1$ inputra 52
6.3. Algorithm-DB: igényvezérelt algoritmus hátramutató szeletekhez 54
6.4. Példa: Algorithm-DB egy lefutása 55
6.5. Algorithm-DF: igényvezérelt algoritmus előremutató szeletekhez 56
6.6. Példa: Algorithm-DF egy lefutása 56
6.7. Algorithm-GB: globális algoritmus hátramutató szeletekhez 58
6.8. Példa: Algorithm-GB egy lefutása 59
6.9. Algorithm-GF: globális algoritmus előremutató szeletekhez 60
6.10. Példa: Algorithm-GF egy lefutása 61
6.11. Algorithm-PB: párhuzamos algoritmus hátramutató szeletekhez 63
6.12. Példa: Algorithm-PB egy lefutása 65

6.13. Algorithm-PF: párhuzamos algoritmus előremutató szeletekhez 66
6.14. Példa: Algorithm-PF egy lefutása 67

7.1. Példa: egy C példaprogram és (statikus) $d : U$ reprezentációja 86
7.2. DynamicSliceForC algoritmus: C programok szeletelésére 88
7.3. Példa: DynamicSliceForC algoritmus egy lefutása 89
7.4. Példa: az $\langle a = 2, b = 6 \rangle$ inputra számított szeletek 90

8.1. Halmazok mérete és halmazműveletek száma 102
8.2. Maximális halmazméretek a feldolgozottság függvényében 103
8.3. Maximális halmazméretek és elemszámok programonként 103
8.4. Uniós szeletméretek hisztogramjai 108
8.5. Monoton szeletméret grafikon uniós szeletekhez 109
8.6. Lefedettség és átlagos uniós szeletméret növekedés 109
8.7. Hátra- és előremutató uniós szeletek méretnövekedése 110

Táblázatok jegyzéke

1.1. Eredmények és publikációk kapcsolata 6

4.1. Példa: minimális d értékek az EA két komponenséhez 25
4.2. Példa: DFC értékek 25
4.3. Hatáshalmaz méretek ER relációhoz 36

6.1. Dinamikus szeletelő algoritmusok áttekintése 49

7.1. A szeletelő algoritmusok megvalósított implementációi 73

8.1. Dinamikus szeletelő algoritmusok számítási komplexitása 94
8.2. Dinamikus szeletelő algoritmusok tárigény komplexitása 94
8.3. C tesztprogramok tulajdonságai 100
8.4. Változók száma 101
8.5. Iterációs számok 104
8.6. A *worklist* mérete 104
8.7. Java tesztprogramok jellemzői 105
8.8. Végrehajtott utasítások 106
8.9. Statikus szeletméretek, adatok 107
8.10. Dinamikus szeletméretek, adatok 107
8.11. Uniós szeletméretek, adatok 108

– Családomnak –

1. fejezet

Bevezetés

A programok analízise egy igen szerteágazó kutatási terület még akkor is, ha a programanalízis fogalmát – egy szűkebb értelmezést véve – korlátozzuk a *programelemek kapcsolatainak felderítésére*. Ez utóbbi olyan – önmagukban is nagy – témaköröket foglal magába, mint a *hatásanalízis* vagy a *programszeletelés*, célszerű tehát ezt a fogalmat is valamilyen módon tovább bontani. Ez a particionálás több dimenzió mentén történhet. Egy ilyen dimenzió lehet az, hogy a programelemek között a program egy konkrét végrehajtása során keletkező függőségeket keressük, vagy az összes (az adott kontextusban elképzelhető) függőséget szeretnénk megtalálni konkrét végrehajtás nélkül. Az első esetben *dinamikus*, a másodikban *statikus* függőségekről beszélünk. Egy következő dimenzió mentén a programelemek szintjei között mozoghatunk. Beszélhetünk *alacsony szintű* analízisről, amikor utasítások vagy alapblokkok között keresünk kapcsolatot, de ugyanúgy meghatározhatóak relációk eljárások, függvények, metódusok, osztályok illetve modulok között is *magas szinten*. Gyakorlati szempontból nem mindegy továbbá az sem, hogy éppen milyen program reprezentációból kell kinyerni a megfelelő adatokat. A megvalósítások dolgozhatnak *bináris kódon*, *forráskódon* de akár magasabb szintű reprezentáción, például *UML* diagramokon is. Ezen felosztások kombinációi különféle megoldásokat igényelnek, és különféle alkalmazási lehetőségeik/területeik vannak.

Munkám során ennek a nagy témakörnek leginkább két részét érintettem: a példaként már említett magas szintű *hatásanalízis*t és alacsony szintű *programszeletelést*. Előbbinél statikus és dinamikus technikákkal is foglalkoztam, míg utóbbi esetében csak dinamikus technikákkal.

A *Hatásanalízis* (*Impact Analysis*) – pontosabban a hatásanalízis alatt leginkább értett *Változás-Hatásanalízis* (*Change Impact Analysis*) – nem más, mint „a változások lehetséges következményeinek beazonosítása, vagy annak megbecslése, hogy egy változás elvégzéséhez mit kell módosítani” [15]. Azt keressük tehát, hogy ha a program egy bizo-

nyos pontján valamit megváltoztatunk, akkor a program mely más pontjait kell legalább átnézni – de elképzelhető, hogy módosítani is – ahhoz, hogy a változtatással együtt a program továbbra is megfelelően működjön. A hatásanalízisnek számos gyakorlati alkalmazása van, például teszteset-szelekció regressziós teszt esetén [52] (hogy csak egyet említsünk), amikor egy apró komponens megváltozása miatt nem lenne hatékony az összes tesztet futtatni, a komponens tesztelése viszont önmagában nem megnyugtató. Ilyenkor hatásanalízis segítségével meg lehet határozni a változtatott komponenshez közvetlenül kapcsolódó egyéb komponenseket, és azokat is tesztelni kell.

A hatásanalízis több szempontból is magas szintű elemzésnek tekinthető. Egyrészt legalább függvény/metódus szintű programelemek közötti kapcsolatokkal operál, másrészt forráskódból vagy magasabb szintű tervezési dokumentációból dolgozik.

A *Programszeletelés* eredeti definícióját Mark Weiser adta meg [59] azon utasítások összességeként, amik hatással lehetnek a program egy szeletelési kritériummal meghatározott viselkedésére. A szeletelési kritérium egy programsor-változó páros, az ehhez tartozó szelet pedig egy olyan futtatható része a programnak, aminek a viselkedése a kritérium tekintetében megegyezik a teljes program viselkedésével. Azaz az adott sorban található változónak a teljes programmal és a szelettel számított értéke minden esetben megegyezik egymással. Ezzel Weiser tulajdonképpen a hátramutató futtatható statikus szeleteket definiálta. Ezen túl persze léteznek előremutató, nem (feltétlenül) futtatható és dinamikus szeletek is, illetve ezen és egyéb jellemzők kombinációjával még rengetegféle programszelet. A különféle szeleteket sokféle tevékenység során lehet alkalmazni, például nyomkövetéskor, párhuzamosításhoz, tesztelésnél, karbantartásnál [56].

A programszeletelés tipikusan alacsony szintű analízis. Egyrészt utasítás szintű kapcsolatokkal dolgozik, másrészt (talán pont ebből kifolyólag) forráskódnál magasabb szintű reprezentációt nem igazán lehet szeletelni, létezik viszont bináris szeletelés [38].

Értekezésemben ezen területeken végzett tevékenységemet és az elért eredményeket foglalom össze.

1.1. Eredmények összegzése

Az elért eredmények a fentiek szerint két nagy csoportra oszthatók. A fenti felbontás szerint a disszertáció első részében a magas szintű analízis (hatásanalízis) területén elért eredményeinket foglalom össze, míg a második részében az alacsony szintű analízis (programszeletelés) témakörében elért eredményeinket.

1.1.1. Magas szintű analízis

A magas szintű analízis az elért eredményeink esetében az eljárás szintű analízist jelenti. Nevezetesen, két olyan eljárások/függvények/metódusok közötti kapcsolatot határoztunk meg, melyek kis költséggel számolhatóak az adott szintű programelemek között, és felhasználhatóak hatásanalízishez, például teszteset-szelekcióhoz. Az egyik a statikusan számítható *Static Execute After/Before* (SEA/SEB) páros reláció, a másik a dinamikus kapcsolatok alapján számolható *Dynamic Function Coupling* (DFC) metrika. Az elért eredményeink a fenti relációk tekintetében a következők:

I/1. *SEA/SEB* relációk definíciója. Statikus környezetre alkalmazva Taweesup Apiwattanapong és társai dinamikus környezetben elért eredményeit [5], definiáltuk a *Static Execute After* (SEA) és *Static Execute Before* (SEB) relációkat és hatékony kiszámítási módjukat. A kísérleti méréseink azt mutatták, hogy az eljárások között definiált relációk megfelelő pontossággal közelítik a statikus szeletelés eljárásokra vetített eredményeit, de sokkal kisebb erőforrást igényel a kiszámításuk. Ráadásul tapasztalataink alapján (természetéből adódóan nem meglepő módon) nemcsak a programelemek közötti közvetlen, hanem a közvetett kapcsolatok kimutatására is alkalmas.

Saját eredmények. A *SEA/SEB* relációk definíciója, illetve a számításukhoz használt *ICCFG* meghatározása szerzőtársaimmal közös eredmény. Az elért eredményeket a [BGJ07] cikkben publikáltuk.

I/2. *DCF* metrika definíciója, meghatározása és kiértékelése. A statikus elemzések hatékonysága bizonyos alkalmazások esetén megkérdőjelezhető, lévén statikusan minél teljesebb eredményt adni csak a dinamikus elemek közötti kapcsolatok konzervatív közelítésével lehetséges, ami nagyon gyakran a kapcsolatok számának nagymértékű túlbecsléséhez vezet. Ezért szerzőtársaimmal bevezettük az eljárások szintjén értelmezett *Dynamic Function Coupling* (DFC) metrikát, melynek segítségével a statikus analízis eredményeinél pontosabb képet kapunk a programelemek közötti kapcsolatokról. A metrika ráadásul paraméterezhető, így a pontosság és a teljesség közötti hangsúlyt az alkalmazás függvényében változtathatjuk. A metrika alapján különféle hatáshalmazok számolhatók. Kísérleti eredményeink alapján elmondható, hogy megfelelő paraméterezéssel duplájára növelhető a kiszámított hatáshalmazok pontossága.

Saját eredmények. A *DCF* metrika definíciója és kiszámításának elve, valamint a metrika mérésekkel történő kiértékelése szerzőtársaimmal közös eredmény. A *DFC* metrikát és a metrikán alapuló hatáshalmazokat számító algoritmusok kidolgozása saját eredmény. Az elért eredményeket a [BGF07] cikkben publikáltuk.

1.1.2. Alacsony szintű analízis

Az alacsony szintű analízis az elért eredményeink esetében a forráskód alapú dinamikus programszeletelést jelenti. Nevezetesen, a szerzőtársaim által kidolgozott két (egy globális illetve egy igényvezérelt) $d : U$ (*definition-use*, (változó) *definíció-használat*) alapú algoritmus mellé további hasonló elven működő szeletelő algoritmusokat definiáltunk. Valamint betömtük az elmélet és a gyakorlat közötti rést, amennyiben az elvi algoritmusok implementálása során a C és Java programozási nyelv forráskód alapú szeletelése során felmerült problémákra megoldásokat adtunk.

A programszeletelés témakörében az elért eredmények a következők:

II/1. *d:U* alapú szeletelő algoritmusok meghatározása. Szerzőtársaim korábbi szeletelő algoritmusaira ([28, 9], [BGS01]) alapozva rendszereztük a lehetséges $d : U$ függőség alapú szeletelési algoritmusokat. A kategorizálás eredményeként a korábban már ismertetett két algoritmus mellé további 6 lehetséges algoritmust vázoltunk fel, melyek közül négyet megvalósításra érdemesnek találtunk és kidolgoztunk. Az algoritmusokról megmutattuk, hogy azok azonos eredményt adnak, mint a *Dinamikus Függőségi Gráfon* (DDG) alapuló széles körben elfogadott algoritmus [3].

Saját eredmények. Az algoritmusok osztályozása, négy új algoritmus (igényvezérelt előremutató előrehaladó feldolgozással, globális előremutató előrehaladó és fordított feldolgozással, globális visszamutató fordított feldolgozással) meghatározása és kidolgozása szerzőtársaimmal közös eredmény. Az algoritmusok *DDG* algoritmussal való ekvivalenciájának igazolásában a saját hozzájárulásom a meghatározó. Az eredményeket a [BGG06] cikkben és az ezt kiegészítő [BGG07] jelentésben publikáltuk.

II/2. *d:U* alapú szeletelő algoritmusok implementációja. A $d : U$ alapú szeletelő algoritmusok – mint általában a legtöbb hasonló elvi megoldás – egy egyszerű, koncepcionális programozási nyelvre lettek definiálva, amelyen a működési elvük demonstrálható, helyességük elméletben igazolható. Viszont már a legegyszerűbb

valós programozási nyelvekben is felmerülnek olyan konstrukciók, amelyeket az elvi megoldás nem pontosan fed le, az elterjedt programozási nyelvek pedig bővelkednek ilyen megoldásokban.

Valós nyelvek forráskód-alapú dinamikus szeletelésében a legtöbb probléma a forráskód és a lefordított és futó bináris kód elemeinek egymáshoz rendeléséből adódik. Mármint abból, hogy ezt hogyan kell vagy érdemes elvégezni.

Az egyik nyelv, amelynek dinamikus szeletelési problémáira megoldást adtunk, a C nyelv. Erre a nyelvre szerzőtársaimmal együtt megoldottuk többek között a mutatók kezelését, a változók és memóriacímek összerendelését, a függvényhívásokon keresztüli adatáramlás követését, a strukturálatlan vezérlési utasítások vezérlési függőségeinek problémáját.

Saját eredmények. Az algoritmusok C nyelvű programokat szeletelő változatának kidolgozása és implementációja során a változók kezelését, azaz a forráskódban látható hivatkozások és a futás közbeni memóriacímek összerendelésének problémáját oldottam meg. Az eredményeket a [BGS01] és [FG02] cikkekben publikáltuk.

II/3. *d:U* alapú szeletelő algoritmusok kiértékelése. A fent említett algoritmusokat többféleképpen is kiértékeltük. Egyrészt az összes algoritmusra meghatároztuk azok számítási- és tárhely-komplexitását. Másrészt egyes algoritmusokat implementáltunk C illetve Java nyelvekre, és valós környezetekben méréseket végeztünk velük az előállított szeletek méretére vonatkozólag.

Saját eredmények. Az elméleti algoritmusok elvi kiértékelése saját eredmény, melyet összefoglaló jelleggel a [BGG06] cikkben, részletesebben a [BGG07] jelentésben publikáltuk. Gyakorlati kiértékeléseket többször végeztünk. A C nyelvű implementáció szeleteinek kiértékelése közös eredmény, amit a [BGS01], [FG02] cikkekben és [BGG07] jelentésben publikáltunk. A Java nyelvű implementációval történt mérések kiértékelése szintén közös eredmény, mely az [SGB07] cikkben lett publikálva.

Az 1.1. ábrán az áttekinthetőség kedvéért összefoglalom, hogy mely eredményeket mely publikációkban publikáltuk.

1.2. A dolgozat felépítése

A dolgozatom a fenti felbontáshoz igazodva két részből áll.

	[BGS01]	[FG02]	[BGG06]	[BGF07]	[SGB07]	[BGJ07]
I/1: *SEA/SEB* relációk						•
I/2: *DCF* metrika				•		
II/1: *d:U* alapú szeletelő algoritmusok meghatározása			•			
II/2: *d:U* alapú szeletelő algoritmusok implementációja	•	•				
II/3: *d:U* alapú szeletelő algoritmusok kiértékelése	•		•		•	

1.1. táblázat. Eredmények és publikációk kapcsolata

I. rész

Az első részben a magas szintű analízissel foglalkozom. A 2. fejezetben bemutatom a hatásanalízis problémáját és rövid áttekintést adok arról, hogy milyen eredmények születtek ezzel kapcsolatban, különös tekintettel az általunk végzett kutatások előzményeire. A 3. fejezetben ismertetem a *SEA/SEB* relációkat, és a meghatározásukhoz használt *ICCFG* gráfot. Ezek után a 4. fejezetben definiálom a *DFC* metrikát, a metrikaértékek számítására szolgáló algoritmusokat, valamint a metrikával kapcsolatos méréseink eredményeit.

II. rész

A disszertációm második részében az alacsony szintű analízissel, konkrétan a programszeleteléssel foglalkozom. Az 5. fejezetben röviden bemutatom a programszeletelés problémakörét és a rá adott megoldásokat. A 6. fejezetben ismertetem a $d : U$ alapú szeletelést és a szeletelő algoritmusainkat, megmutatom, hogy a kapott eredményeik megegyeznek az egyfajta etalonnak számító *DDG* alapú szeletelés eredményeivel, majd a a 7. fejezetben az algoritmusok C nyelvre történő implementációjának problémáiról, és azok egy részének megoldásáról írok. A 8. fejezetben az algoritmusok kiértékelése történik, elméleti és gyakorlati alapokon.

Végül a 9. fejezetben még egyszer összefoglalom az elért eredményeket. Dolgozatom végén az A és B függelékekben a teljes dolgozat magyar illetve angol nyelvű összefoglalója található.

I. rész

Magas szintű analízis

2. fejezet

Hatásanalízis

Ebben a fejezetben összefoglaló jelleggel bemutatjuk, hogy mi is az a hatásanalízis, milyen tudományos eredmények születtek már vele kapcsolatban, illetve mire lehet használni.

2.1. Hatásanalízis

Számos, a szoftver evolúcióhoz kapcsolódó szoftverfejlesztési tevékenység során csak a szoftverrendszer bizonyos részeit vizsgáljuk egyszerre, és ez a számunkra érdekes rész bővül vagy változik a tevékenységünk során. Nevezetesen, az inkrementális változásokon (Incremental Change) [49] alapuló szoftver életciklusban meg kell határozni a változásoknak a rendszerre gyakorolt hatását, amit azután felhasználhatunk változás kiterjesztéshez (change propagation), regressziós teszteléshez és egyéb tevékenységekhez. Ha például elvégzünk egy ilyen inkrementális változást a szoftveren, az általában úgy történik, hogy először meghatározzuk, hogy a program mely pontjait (pl. osztályait, metódusait) érinti a változtatás lényege, majd ezekből a pontokból kiindulva a változtatás lehetséges kihatásai mentén haladva a változtatást a teljes rendszeren keresztül végigvisszük. Ilyenkor, miután a program egy részét sikeresen megváltoztattuk, a változtatott rész szomszédos elemeire koncentrálunk, egészen addig, míg tulajdonképpen meg nem határozzuk a kezdeti változtatás hatáshalmazát (Impact Set). Vagy egy másik példaként említhetjük a regressziós tesztelést [52], amikor is egy változtatás nyomán a teljes rendszer tesztelése helyett – például erőforráshiány miatt – csak bizonyos megfelelően kiválasztott teszteseteket hajtunk újra végre. Az ilyen módon újratesztelendő elemek most is a megváltozott részek és azok szomszédsága alapján határozhatóak meg.

A „szomszédság” meghatározása persze erőteljesen függhet attól, hogy éppen milyen célból is keressük ezeket. Ha például *változás hatásanalízis* (Change Impact Analysis)

során egy változtatás miatt átnézendő programelemeket keresünk, a szomszédság praktikusan szimmetrikus lesz, míg teszteset kiválasztáshoz *számítási hatásanalízist* (Computational Impact Analysis) célszerű használni, ami a végrehajtási sorrendet is figyelembe veszi, így nem feltétlenül szimmetrikus.

A legegyszerűbb statikus módszerek a hívási gráfot [15] vagy valamilyen más könnyűsúlyú (kis számításigényű) függőséget használnak erre a célra, ami pontatlan[1] vagy nem biztonságos[2] eredményhez vezet (pl. [64]). Lehet természetesen (bizonyos mértékig) pontos és egyben biztonságos módszereket és eredményeket is találni, mint ahogyan például a statikus programszeletelést is javasolták, de ennek a számításigénye túl magasnak bizonyult [30, 56] a feladathoz.

A hatásanalízist segítő relációk meghatározására különböző megközelítések léteznek [7], de közös jellemzőjük, hogy akármilyen részletességgel számítjuk is a függőségeket, az eredményeket mindig magasabb szintű szoftverelemek között értelmezzük. A legtöbb általános hatásanalízis módszer statikus, mint például Rajlich vagy Ren és társaik munkái [49, 50]. Rajlich és Gosavi definiáltak egy életciklus-modellt, ami erőteljesen támaszkodik a statikus változás hatásanalízisre, míg Buckner és társai kifejlesztettek ehhez egy kísérleti eszközt [19]. Tonella és társai egy program megértést segítő változó pontosságú algoritmust mutattak be az egyes programelemek egymásból való elérhetőségének meghatározására [57]. Léteznek, sőt, viszonylag gyakoriak a konkrét alkalmazási környezetben használható módszerek is. Például kifejezetten regressziós teszteléshez, pontosabban szelektív újratesztelésehez léteznek speciális módszerek (Rothermel és Harrold készített egy nagyszerű összehasonlítást teszt szelekciós technikák között [52]). Rothermel és Harrold például vezérlési folyam gráfot használ biztonságos teszt szelekcióhoz [53], de statikus módszereket használt Binkley [12] valamint Gupta és társai [27] is. Bizonyos módszerek pedig különösképpen használhatónak bizonyultak mert bár kevésbé precízek, nagyon hatékonyak, mint például a *firewall testing* [60, 61]. A legtöbb ilyen módszer szintén statikus analízisen alapul [15].

De a statikus algoritmusoknál még egy precíz, szeletelés alapú megközelítés esetén is nagyon valószínű, hogy a hatáshalmazok mérete túl nagy lesz ahhoz, hogy valódi segítséget jelentsen az aktuális problémánk megoldásában. Ennek oka a statikus algoritmusok eredendő konzervativizmusa például a változó aliasinggal, dinamikus polimorfizmussal

[1] Pontos az a módszer/eredmény, ami egyetlen egy valójában soha nem realizálódó – tehát felesleges – függőséget sem hoz be. A legtriviálisabb ilyen eredmény az, amikor nincsenek szomszédaink, azaz minden elemet önmagában tekintünk.

[2] Biztonságosnak azt a módszert/eredményt nevezzük, amely egyetlen egy potenciálisan realizálható függőséget sem hagy ki. A legtriviálisabb biztonságos függőség, amikor azt mondjuk, hogy a rendszer minden eleme a rendszer minden más elemével összefügg, ez viszont eléggé nyilvánvalóan egy gyakorlatilag használhatatlan eredmény.

vagy kivételkezeléssel szemben. Ezért van létjogosultsága a dinamikus algoritmusoknak, ahol ezeket a problémákat nem a legrosszabb eshetőségre felkészülve kell kezelnünk, hanem tulajdonképpen a problémák nem is léteznek, mert a futás során a megfelelő információ rendelkezésünkre áll. Bár a legpontosabb megoldás nyilvánvalóan egy részletes dinamikus programszeletelés [3, 40] eredménye lenne, a nagy számításigény miatt ezt nem éri meg hatásanalízisre használni. Így aztán a kevésbé részletes, de jóval hatékonyabb – például dinamikus hívási gráfon alapuló – algoritmusok használhatóbb alternatívát nyújtanak. Valójában a jelenlegi állás szerint csak függvényszintű dinamikus analízis használható éles körülmények között valódi szoftverrendszerek valódi tesztesetein [42, 48].

Az egyik leghasználhatóbb ilyen módszer eljárásszintű dinamikus hatáshalmazok megadására a függvény-belépési/kilépési eseményeket tartalmazó végrehajtási nyomokból előállítható úgynevezett *Execute After* sorozatok számítása [5]. Apiwattanapong és társai egy nagyon egyszerű megközelítést alkalmaztak, amely lényegében így hangzik: véve egy sor futtatást, az f függvénynek potenciálisan hatása van minden olyan metódusra, amit valamikor az f után hajtottunk végre, vagyis minden olyan g függvény, amit f után futtattunk, benne van az f hatáshalmazában. A hatáshalmaz nagyon könnyen számítható, egész egyszerűen a végrehajtási nyomban előforduló legelső f bejegyzés után szereplő összes függvényt tartalmazza. Bár az eljárás biztonságos – azaz nem hagy ki egyetlen valódi függőséget sem –, meglehetősen pontatlan. Viszont pusztán a függvényhívási információkra támaszkodva lehetetlennek látszik egy ennél pontosabb, de mégis biztonságos módszer megadása.

Persze azért további módszerek is léteznek. Orso és társai egyfajta közelítő statikus szeletelés és a dinamikus lefedettségi információ kombinációját ajánlották hatásanalízisbeli alkalmazásra [48]. A statikus rész tartalmaz egy CFG-n dolgozó elérhetőségi algoritmust is, a dinamikus rész pedig egyszerű és hatékony, de figyelmen kívül hagyja a függvények hívási sorrendjét. Law és Rothermel is bemutatott egy függvényszintű dinamikus algoritmust [42], amely a végrehajtási nyom feldolgozásán alapszik. Bár precízebb, hely- és tárigénye magasabb (többek között a nyom hosszától is függ). Breech és társai hatásanalízis módszere a dinamikus hívási gráfon dolgozik, és a hívási és visszatérési láncok alapján számol hatáshalmazokat [16].

Hatásanalízis-számításhoz használhatóak különféle csatolási metrikák is, amelyek erősebb kapcsolatok esetén magasabb értékeket vesznek fel. Számos ilyen metrikát javasoltak és értékeltek ki például Briand és társai [17, 18]. Arisholm és társai objektumorientált rendszereken értelmezett dinamikus csatolási metrikákat értékeltek ki [6]. Mitchell és Power pedig statikus és dinamikus csatolási metrikák kapcsolatát vizsgálta [44]. Azt

találták, hogy önmagukban a statikus metrikák nem minden esetben megfelelőek. Yang és társai pedig indirekt csatolások vizsgálatával foglalkoztak [63].

3. fejezet

A Static Execute After/Before (*SEA/SEB*) relációk

Ebben a fejezetben bemutatjuk az általunk definiált *Static Execute After/Static Execute Before* (SEA/SEB) relációkat. A relációk definíciója és a számításukhoz használt *Interprocedural Component Control Flow Graph* (ICCFG) meghatározása szerzőtársaimmal közös eredmény, a 3.4. fejezetben bemutatott eredmények elérésében Jász Judit szerzőtársam munkája volt meghatározó.

3.1. Motiváció

Mint azt az előző fejezetben láthattuk, a hatáshalmazok számítása azon alapszik, hogy hogyan határozzuk meg a programelemek közötti közvetlen kapcsolatokat. A komponensek közötti kapcsolatok egy része (mint például objektumorientált rendszerekben az öröklődés, kompozíció, asszociáció, stb.) explicit, azaz a függő komponensek kölcsönösen ismerik egymást, expliciten megjelenhetnek egymás kódjában is. Ezek mellett azonban léteznek az úgynevezett *rejtett függőségek*[1] is, amelyek esetében a függő komponensek nem feltétlenül tudnak egymásról. Ilyenek lehetnek például a Rajlich és társai [64] által kifejtett, adatfolyamokon keresztül közvetített rejtett függőségek. Ők a rendszerfüggőségi gráfokon (*System Dependence Graph* – SDG [30]) alapuló absztrakt rendszerfüggőségi gráfokat (Abstract System Dependence Graph – ASDG) használták, de nagy programok esetén erőforrásigényük miatt ezek használhatatlanok.

Az programkomponensek közötti explicit és rejtett függőségek meghatározására ve-

[1] A „rejtett” jelző itt a hatásanalízisbeli viszonylagos rejtettséget jelöli. Hatásanalízis esetében ugyanis gyors, kevés számítást igénylő megoldásokat használunk. Egy részletesebb analízis (például programszeletelés) ezeket a rejtettnek nevezett függőségeket is képes lenne a felszínre hozni, de a költségei miatt ilyen mélyebb elemzéseket a hatásanalízisben nem végzünk.

zettük be a *Static Execute After* (SEA) és *Static Execute Before* (SEB) relációkat. A megközelítésünket az Apiwattanapong és társai által bevezetett *Execute After* (EA) reláció [5] motiválta. Az EA relációt ők dinamikus környezetben értelmezték, a SEA ennek a statikus párja. A relációkban egy erősen konzervatív megközelítés ölt testet, így a reláció ugyan minden lehetséges függőséget tartalmaz (azaz biztonságos), de sok hamis függőség is a része lesz (pontatlan). A reláció lényege egyszerűen az, hogy egy g eljárást akkor tekintünk egy f eljárástól függőnek, ha az összes lehetséges végrehajtási utat figyelembe véve a g eljárást „végrehajthatjuk" az f eljárás után.

Ugyanezzel a logikával természetesen akár osztályok közötti SEA relációról is beszélhetünk, vagyis egy A osztályt akkor tekintünk egy B osztálytól függőnek, ha az A osztályt „végrehajthatjuk" a B osztály után. Nyilvánvaló, hogy ez a definíció szükségszerűen lefed minden, az A és B osztályok között létrejövő kapcsolatot (az adatfolyam alapúakat is beleértve), és ami még fontosabb, a szintaktikusan akár fel sem fedezhető szemantikus kapcsolatokat is (mint például a program két pontján ugyanannak a konstans értéknek a használatán alapuló szemantikus kapcsolat). Továbbá az egyszerű reláció egyszerű és hatékony algoritmusok kidolgozását teszi lehetővé, legalábbis a részletes, de épp ezért pontosabb adatfolyam alapú módszerekhez képest.

A témához kapcsolódó publikációkban különféle megközelítések léteznek a hatásanalízist segítő, magasabb szintű szoftverelemek közötti relációk meghatározására [7]. Mi egy eljárások szintjén dolgozó, csak folyamat-relációkat használó egyszerűsített megközelítést választottunk, aminek az alapja a gráfpont-elérés számítás és a függvényen belüli vezérlési folyam gráf. Tonella és társai is bemutattak egy program megértést segítő változó pontosságú utasításszintű algoritmust az elérhetőség meghatározására [57]. Orso és társai egyfajta (dinamikus lefedettséggel kombinált) közelítő statikus szeletelést ajánlottak hatásanalízisbeli alkalmazásra [48]. Ebben részletek nélkül említik egy CFG-n dolgozó elérhetőségi algoritmus használatát.

Hasonlóképpen a relációkhoz, a programábrázolás módjára is igen sokféle megoldás található a kapcsolódó irodalomban. A *Context Sensitive Control Flow Graph* (CSCFG) leginkább vizualizációs célokra lett használva Ng által [47]. Ez annyiban hasonlít a mi programábrázolásunkra, hogy szintén a CFG-ből indul és a lényegtelen részeket a pontok összevonásával – a lényeges pontok vezérlési folyam információinak megtartása mellett – eltünteti.

3.1.1. A SEA lehetséges alkalmazásai

A SEA/SEB relációk egyik alkalmazása a lehetséges rejtett függőségek meghatározása. A rejtett függőségekre egy egyszerű példa látható a 3.1. ábrán. Bár adatfolyam-

analízissel felfedezhető lenne, hogy a B osztály „függ” az A osztálytól, a legtöbb magas szintű hatásanalízis módszer ezt figyelmen kívül hagyja, hiszen az A és B osztályok közvetlenül nem ismerik egymást. Ez tehát számukra rejtett függőség.

```
class C {
  ...
  A a;
  B b;
  ...
  void foo() {
    ...
    b.paint(a.get());
    ...
  }
};
```

3.1. ábra. Példa: rejtett függőség

A SEA reláció nagyon egyszerűen használható a rejtett függőségek megbecslésére. Az `a.get()` hívását a `b.paint()` hívása követi, ezért a SEA alapján a B osztály függ az A osztálytól. És mivel ezt a függőséget a szokásos hatásanalízis módszerekkel explicit módon nem fedezhetjük fel, szükségképpen rejtett függőség lesz.

A rejtett függőségek azután felhasználhatóak egyéb területeken, mint például hiba kiterjesztéshez (*change propagation*). Amikor egy programrészlet megváltozik, a változás jellegéből fakadóan előfordulhat, hogy többé nem illeszkedik megfelelően a környezetéhez (például egy értéket a környezetétől eltérően kezd értelmezni). A fejlesztő feladata tehát, hogy az itt eszközölt változást végigvezesse a programrendszer egészén. Ehhez nyilvánvaló segítség, ha valaki vagy valami meg tudja mondani, hogy mik a változott kódrészlet „szomszédai”. Ezeket végigvizsgálva eldönthető, hogy azokba kell-e újabb változtatásokat belevinni. Ha igen, akkor ezek szomszédait is vizsgálni kell, és így iteratívan előbb-utóbb minden, a változásban potenciálisan érintett kódrészlet át lesz vizsgálva.

Nagyon fontos tehát a „szomszédság” fogalom meghatározása, ha ugyanis valamilyen kapcsolatról elfelejtkezünk, akkor a rendszerben olyan kód maradhat, ami a változtatások révén elavult, ezáltal pedig a program hibásan fog működni. A rejtett függőségek ilyen szempontból különösen veszélyesek, hiszen általában sokkal nehezebben felfedezhetőek, mint az explicit függőségek, ezáltal sokkal valószínűbb, hogy rejtett hibák forrásaivá válnak. Szükséges tehát – többek között – a rejtett függőségeket is beleérteni a szomszédsági relációba. A SEA így alkalmas lehet a szomszédság meghatározására, mivel a rejtett függőségeket is felderíti.

A változott programkódot természetesen ezután még tesztelni kell. Az egyik legfontosabb talán a rendszer regressziós tesztelése, ami azt ellenőrzi, hogy a változás hatására a program hibásan kezd-e működni valamely már ismert teszteseten, amire eddig helyesen működött. A legegyszerűbb megoldás természetesen az összes teszteset újrafuttatása, ez viszont – főleg nagy rendszereknél – nem járható út. Így a tesztesetek közül valamilyen módon szelektálni kell. A tűzfal tesztelés (firewall testing) [60] egy heurisztikus módszer, amely a változott programkomponenseket és azok közvetlen szomszédait válogatja ki újratesztelésre. A szomszédság meghatározására – csakúgy, mint az előző esetben – érdemes lehet a rejtett függőségeket is tartalmazó SEA relációt használni.

3.2. A *SEA/SEB* relációk

Apiwattanapong és társai [5] definiálták az *Execute After* (EA) relációt, ami akkor áll fenn két függvény között, ha valamely részeik egymás után futnak. Pontosabban (definíció szerint), ha az egyik (tranzitíven) hívja a másikat, visszatér belé vagy valamilyen közös „ősből" először az egyik, majd ezután a másik függvény hívódik meg (tranzitíven). Az EA dinamikus relációként mindig a program egy vagy több konkrét futására vonatkozik, tehát az adott végrehajtás(ok) során fellépő relációkat tartalmazza.

Ezen reláció statikus párjaként definiáljuk a *Static Execute After* (SEA) relációt. Azt mondjuk, hogy $(f, g) \in SEA$ akkor és csak akkor, ha g bármely része potenciálisan végrehajtódhat, miután f valamelyik részét végrehajtottuk. Nyilvánvaló, hogy a SEA tartalmazza a részletes rendszerfüggőségi gráfon (SDG) számolt eljárásszintű relációkat [30] (vagyis a statikus szeletelés eredményének eljárásszintre emelését). Vagyis a SEA egy biztonságos reláció, ám meglehetősen pontatlan: sok hamis függőséget is behoz.

Formálisan a SEA/SEB relációk definícióját a fentebb említett hármas felbontás szerint adjuk meg. Vagyis, definíció szerint

$$SEA = SEA_{call} \cup SEA_{seq} \cup SEA_{ret} \ ,$$

ahol

$$\begin{aligned}
(f, g) \in SEA_{call} &\overset{\text{def}}{\iff} f \text{ hívja } g\text{-t}, \\
(f, g) \in SEA_{seq} &\overset{\text{def}}{\iff} \exists\, h\text{: először } h \text{ hívja } f\text{-et, majd} \\
&\qquad \text{miután } f \text{ visszatért } h\text{-ba, } h \text{ hívja } g\text{-t}, \\
(f, g) \in SEA_{ret} &\overset{\text{def}}{\iff} f \text{ visszatér } g\text{-be},
\end{aligned}$$

ahol mind a „hívja”, mind a „visszatér” tranzitíven értendő. Hasonlóképpen definiálható a SEB reláció is:

$$SEB = SEB_{call} \cup SEB_{seq} \cup SEB_{ret} \ ,$$

ahol

$$\begin{aligned}
(f,g) \in SEB_{call} &\overset{\text{def}}{\iff} g \text{ hívja } f\text{-et}, \\
(f,g) \in SEB_{seq} &\overset{\text{def}}{\iff} \exists\, h\text{: először } h \text{ hívja } g\text{-t, majd} \\
&\qquad \text{miután } g \text{ visszatért } h\text{-ba, } h \text{ hívja } f\text{-et}, \\
(f,g) \in SEB_{ret} &\overset{\text{def}}{\iff} g \text{ visszatér } f\text{-be.}
\end{aligned}$$

Vegyük észre, hogy a két reláció valójában egy, hiszen $SEA_{call} = SEB_{ret}$ és $SEA_{ret} = SEB_{call}$, és $(f,g) \in SEA_{seq} \iff (g,f) \in SEB_{seq}$. Továbbá szabályos, struktúrált függvényhívások esetén $(f,g) \in SEA_{call} \iff (g,f) \in SEA_{ret}$. Ez nagyban egyszerűsíti a feladatunkat, ugyanis e tények ismeretében elegendő csak a SEA relációval foglalkozni.

3.3. Az ICCFG gráf

A *SEA* reláció kiszámításához szükség van a program megfelelő ábrázolására. A hagyományos hívási gráf (*Call Graph* (CG) [54]) erre nem alkalmas, ugyanis az eljáráson belüli eljáráshívások sorrendjéről semmilyen információt nem tartalmaz. Másrészt viszont az interprocedurális vezérlési folyam gráf (*Interprocedural Control Flow Graph* – ICFG [41]) túl sok információt tartalmaz, így a használata túlságosan drága lenne. Szükség van tehát egy saját reprezentációra.

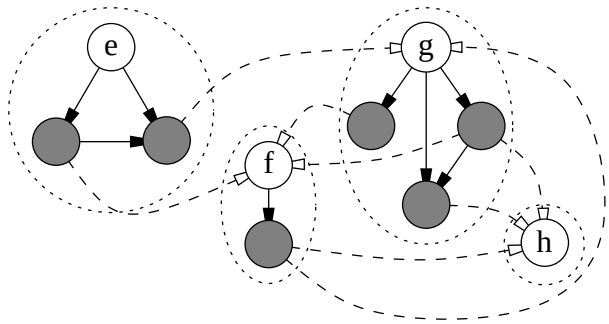
Először a hagyományos vezérlési folyam gráf (CFG) mintájára definiálunk egy intraprocedurális komponens vezérlési folyam gráfot (CCFG) (*Component Control Flow Graph*), amelyben csak az eljáráshívás szempontjából érdekes helyeket és kapcsolatokat tartjuk számon. Minden egyes CCFG egy konkrét eljárást reprezentál, és tartalmaz egy belépési pontot (*entry node*) és számos komponens pontot (*component node*), valamint a pontokat összekötő vezérlési folyam éleket (*control flow edges*). Egy-egy komponens pont egy vagy több eljáráshívást reprezentál oly módon, hogy ha két hívási hely kölcsönösen elérhető egymásból, akkor hozzájuk a CCFG-ben ugyanaz a komponens pont tartozik. A CCFG-t tehát úgy kapjuk a CFG-ből, hogy első lépésben törlünk belőle minden olyan pontot, ami nem eljáráshívás, figyelve arra, hogy az éleket a megfelelő módon módosítsuk. Így egy p és egy q pont között akkor és csak akkor lesz él, ha a p hívási helyről a CFG-ben elérhető a q úgy, hogy közben nem érintünk más hívási helyeket.

```
e() {
  if(...) {
    f();
  }
  g();
}


f() {
  while(...) {
    h();
    g();
  }
}
```

```
g() {
  if(...) {
    f();
  } else {
    while(...) {
      h();
      if(...) {
        f();
      }
    }
    h();
  }
}

h() {
}
```



3.2. ábra. Példa: ICCFG

Továbbá az így kialakult gráfban a szorosan összefüggő komponenseket egyetlen pontba vonjuk össze, így a CCFG egy irányított körmentes gráf lesz. Természetesen az összevont komponenspontokból az összes olyan függvény hívható, amelyet az adott komponenst alkotó függvényhívási helyekről meg lehetett hívni.

Az interprocedurális komponens vezérlési folyam gráf (*Interprocedural Component Control Flow Graph* – ICCFG) az egész rendszert reprezentálja oly módon, hogy minden eljáráshoz tartalmaz egy-egy CCFG-t, amiket hívási élekkel (*call edges*) kötünk össze a megfelelő módon. Vagyis az ICCFG-ben egy c komponens pontból akkor és csak akkor mutat hívási él egy m eljárás belépési pontjára, ha a c pont által reprezentált hívási helyek valamelyike meghívja az m eljárást. A 3.2. ábrán egy példakód és a hozzá tartozó ICCFG látható.

3.4. A SEA kiszámítása

Az ebben a fejezetben leírt eredmények elérésében az elvi alapok közös kidolgozása mellett Jász Judit munkája volt meghatározó.

Az ICCFG alapján a 3.3. ábrán látható algoritmus előállítja a SEA relációt. Az algoritmus kezdetben az eljárások tranzitív híváslistája alapján meghatározza a $SEA_{call} \cup SEA_{ret}$ részrelációt (1-5. sor). Ezután minden eljárásra meghatározza a SEA_{seq} részrelációt. Ehhez először topologikusan rendezi az eljárás komponenseit és a továbbiakban ilyen sorrendben dolgozza fel őket (7-8. sor), így ugyanis semmilyen komponens nem lesz a megelőző komponenseinél korábban feldolgozva. Ez azért fontos, hogy az eljárásból a

komponens előtt hívott eljárásokat megfelelően tudjuk számolni (9., 13-14. sor). Ha egy komponens több eljárást hív, akkor ő egy valamilyen ciklusból összevont pont, így minden eljárása bármelyik eljárása előtt hívódhat (10-11. sor). Végül a SEA bővíthető az olyan (f, g) párokkal, ahol f a komponens előtt, g pedig a komponensből hívódik (12. sor).

```
    program ComputeSEA(P)
    input: P : a program ICCFG reprezentációja
    output: SEA : a program eljárásain értelmezett SEA reláció
    begin
 1:     SEA := ∅
 2:     forall m ∈ P eljárásra do
 3:         transCallsOf[m] := m tranzitív hívásai
 4:         SEA := SEA ∪ ({m} × transCallsOf[m])
 5:         SEA := SEA ∪ (transCallsOf[m] × {m})
        endforall
 6:     forall m ∈ P eljárásra do
 7:         topOrder := componentQueue(m)
 8:         for c := first(topOrder) to last(topOrder) do
 9:             prev[c] := ∅ ∪ ⋃_{p∈c-t megelőző komponensek} prev[p]
10:             if c több eljárást hív közvetlenül then
11:                 prev[c] := prev[c] ∪ transCallsOf[c]
                endif
12:             SEA := SEA ∪ (prev[c] × transCallsOf[c])
13:             if c egyetlen eljárást hív közvetlenül then
14:                 prev[c] := prev[c] ∪ transCallsOf[c]
                endif
            endfor
        endforall
    end
```

3.3. ábra. ComputeSEA algoritmus

Jelölje n az eljárások számát, k és e az eljáráson belüli maximális komponens és élszámot, valamint m az egy komponensből induló hívási élek maximális számát. Ekkor a számítási komplexitás a megfelelő adatstruktúrák használatával a legrosszabb esetben $O(n \cdot e + n \cdot k \cdot m)$, ami nagyjából megegyezik a részletes rendszerfüggőségi gráf (*System Dependence Graph* – SDG) alapú statikus szeletelés [30] komplexitásával. Lényeges különbség viszont, hogy az SDG felépítése bonyolultabb, mérete (eljáráson belüli pontok száma) pedig nagyobb, mint az ICCFG hasonló jellemzői. Bár mindkettő alapja az *Interprocedural Control Flow Graph* (ICFG), míg az ICCFG összefüggő komponensei ebből két bejárással meghatározhatóak és ezáltal csökken a gráf mérete is, addig az SDG-hez szükséges vezérlési és adatfüggőségi élek számítása bonyolultabb algoritmusokon alap-

szik, és nem csökkenti a csúcsok számát sem. Vagyis összességében a SEA olcsóbban számítható.

3.4.1. A SEA javításai

Meg kell említenünk még két olyan dolgot, amivel a SEA használhatóbbá tehető.

Adatfolyam kezelése

A SEA egyik hátránya, hogy csak a függvényhívásokon keresztüli adatáramlást érzékeli. Objektumorientált rendszerek esetében például az adattagok közvetlen írásával/olvasásával olyan kapcsolatokat realizálhatunk, amelyeket hatásanalízisben figyelembe kell venni, de a SEA előtt rejtve maradnak. A probléma azonban megoldható az adatelérések becsomagolásával, azaz ál *getter/setter* metódusok beiktatásával. Valahányszor tehát egy `X` (globális vagy tag-) változót olvasunk, azt az ICCFG-ben egy `getX()` függvényhívással helyettesítjük, a változónak értékadást pedig egy `setX(e)` hívással (ahol `e` az értékadás jobb oldalán szereplő kifejezés).

Pontosság javítása

Két egymás után hívott eljárás között akkor lesz valódi kapcsolat, ha a második (akár csak tranzitíven is) felhasznál valamit az első által előállított adatokból. Ezért bevezetünk négy kategóriát. Egy eljárás akkor adatforrás, ha előállít valamilyen később felhasználható adatot (pl. visszatérési értéket), és akkor adatnyelő, ha felhasznál valami korábbi adatot. Egy tiszta adatforrás *fehér*, egy tiszta adatnyelő *fekete*, ha mindkettő egyben, akkor *szürke*, és ha egyik sem, akkor *átlátszó*. Nyilvánvaló, hogy az $(f, g) \in SEA$ törölhető, ha f fekete vagy átlátszó, vagy g átlátszó vagy fehér.

3.5. Összefoglalás

A fejezetben definiáltuk a függvények között értelmezett *Static Execute After* (SEA) és *Static Execute Before* (SEB) relációkat, valamint megadtuk a kiszámításuk egy módját. Ehhez definiáltuk az *Interprocedural Component Control Flow Graph* (ICCFG) gráfot, ami függvények belépési pontjait és hívási helyeket reprezentáló pontokból áll, hívási és vezérlési folyam élekkel összekapcsolva.

4. fejezet

A Dynamic Function Coupling (*DFC*) metrika

Ebben a fejezetben bemutatjuk az eljárások szintjén értelmezett általunk definiált *Dynamic Function Coupling* (DFC) metrikát, és a metrikát illetve az általa definiált hatáshalmazokat kiszámító algoritmusokat. Bemutatjuk továbbá a metrikával kapcsolatos méréseinket és azok elemzését. A DCF metrika definíciója és kiszámításának elve, valamint a metrika mérésekkel történő kiértékelése szerzőtársaimmal közös eredmény. A DFC metrikát és a metrikán alapuló hatáshalmazokat számító algoritmusok kidolgozásában a saját munkám volt meghatározó.

4.1. Motiváció

Az Apiwattanapong és társai [5] által definiált *Execute After* relációval kapcsolatban két problémát fogalmaztunk meg. Az első, hogy a reláció egyirányú. Ez nem probléma számítási hatásanalízis (*Computation Impact Analysis*) esetén, például regressziós tesztelésnél, amikor a változott kódrészlet miatt újrafuttatandó teszteseteket határozzuk meg. Változás-hatásanalízis (*Change Impact Analysis*) esetén viszont a kapcsolat két elem között szimmetrikus; ha az A elem változása miatt meg kell nézni a B elemet, akkor a B változása esetén is figyelni kell az A elemre [49]. A második probléma, hogy bár a reláció biztonságos, nem túl pontos. Megvizsgáltuk tehát annak a lehetőségét, hogy a jobb használhatóságért némi biztonságot feláldozzunk a nagyobb pontosság érdekében. Ezek mentén azután tovább finomítottuk az EA relációt.

A finomítás alapötlete az, hogy ha két függvény végrehajtása „közelebb” van egymáshoz, akkor érzésünk szerint nagyobb valószínűséggel függenek egymástól. De mit jelentsen a „közelebb”? Szimplán két függvény be- és/vagy kilépési pontjainak a futás során

mért távolsága nem biztos, hogy megfelelő mérőszám, mert így egy közvetlenül f-ből hívott g is távolinak tűnhet, ha előtte és utána is rengeteg eljárás hívódik. Megfelelőbbnek tűnik tehát a dinamikus hívási gráfban távolságokat számolni. Ezek után definiálható a *Dynamic Function Coupling* (DFC) metrika, mint két függvény közötti legkisebb ilyen távolság. Ha pedig megvan a metrikánk, bármely f függvényre kiszámolhatjuk a hozzá egy adott d értéknél közelebb eső függvényeket, amik majd a hatáshalmazt alkotják. Már csak az a kérdés, hogy d értékét meg tudjuk-e, illetve milyen szempontok szerint tudjuk meghatározni. Ennek eldöntését a gyakorlati mérések során szerzett tapasztalatok is segíthetik.

A munkánk részben lefedi Breech és társai dinamikus hatáshalmaz számítási módszerét [16]. A szerzők egy dinamikus hívási gráfon alapuló online algoritmust készítettek hatáshalmazok megadására. A kiszámolt reláció megfelel az általunk használt reláció egy részének.

4.2. A *DFC* metrika

Apiwattanapong és társai definiálták az *Execute After* (EA) relációt [5] egy adott program f és g metódusa között a program egy adott végrehajtása mellett a következő módon[1]: $(f, g) \in EA$ akkor és csak akkor, ha f (direkt vagy tranzitív módon) hívja g-t vagy f (direkt vagy tranzitív módon) visszatér g-be vagy f (direkt vagy tranzitív módon) egy olyan h metódusba tér vissza, ami később (direkt vagy tranzitív módon) meghívja g-t. A DFC metrika megadásánál követjük ezt a metódusok közötti hármas bontást, mert ezáltal is finomíthatunk a relación. A formális definíciók egyszerűsítése érdekében bevezetünk néhány dolgot.

Egy P program T *végrehajtási története* (végrehajtási útja) jelen esetben függvény belépési és függvény kilépési vagy visszatérési események sorozata. Mivel kivétel- és szálkezeléssel most nem foglalkozunk, a végrehajtási nyomban minden belépési eseménynek megvan a megfelelő kilépési párja, és ezek a párok egymásba ágyazva tiszta zárójelezési struktúrát alkotnak. Például: $T = \langle f_e, g_e, h_e, h_r, k_e, g_e, g_r, k_r, l_e, l_r, g_r, k_e, g_e, g_r, f_e, f_r, k_r, f_r \rangle$ egy végrehajtási út, ahol f_e az f függvénybe belépést (***e**ntry*), f_r pedig a függvény elhagyását (***r**eturn*) jelöli.

A *dinamikus hívási fa* a statikus hívási gráf olyan dinamikus megfelelője, ahol egy függvény különböző hívásai meg vannak különböztetve egymástól, és az egy csúcsból

[1] Az egységesség és az általunk használt fogalmakkal, jelölésekkel való konzisztencia kedvéért az eredeti relációt megfordítottuk, és nem foglalkozunk a több végrehajtást magában foglaló verzióval sem.

kiinduló éleknek sorrendje van. Egy P program $T = \langle f_e, T_1, \ldots, T_n, f_r \rangle$ végrehajtási útvonala által definiált dinamikus hívási fa tehát egy olyan G fa, melynek gyökere f-fel van címkézve, a gyökérnek n sorszámozott gyereke van, és azok sorrendben a $T_1, \ldots, T_n$ végrehajtási utaknak megfelelő dinamikus hívási fák. Egy $\langle f_e, f_r \rangle$ végrehajtási úthoz nyilvánvalóan egy f-fel címkézett levél pont tartozik.

Szintén használni fogjuk a G-beli $f \to g$ *hívási lánc* fogalmát, ami egy útvonal p és q pontok között, ahol a p és q rendre f és g függvényekkel címkézett pontok, és a fa gyökerétől p-ig tartó útvonal prefixe a fa gyökerétől q-ig tartó útvonalnak.

Ezek után megadjuk az eredeti *Execute After* (EA) reláció egy d indirekciós paraméter szerinti kiterjesztését. A definíciót a fentebb már említett hármas bontás szerint három részrelációval adjuk meg, amik a függvények egymás utáni végrehajtásának három módját reprezentálják:

$$
\begin{aligned}
(f,g) \in EA^{(d)}_{call} &\overset{\text{def}}{\iff} \exists\, d \text{ hosszúságú } f \to g \text{ hívási lánc,} \\
(f,g) \in EA^{(d)}_{ret} &\overset{\text{def}}{\iff} \exists\, d \text{ hosszúságú } g \to f \text{ hívási lánc,} \\
(f,g) \in EA^{(d)}_{seq} &\overset{\text{def}}{\iff} \exists\, h \text{ metódus, amire:} \\
&\quad \exists\, d_r \text{ hosszúságú } h \to f \text{ és } d_c \text{ hosszúságú } h \to g \text{ hívási lánc} \\
&\quad \text{egyetlen közös } (h \text{ címkéjű}) \text{ ponttal a hívási fában, ahol} \\
&\quad f \text{ előbb van meghívva, mint } g, \text{ és } d = d_r + d_c - 1.
\end{aligned}
$$

Ezek kombinációjával kapjuk azt az $EA^{(d)}$ relációt, amelyik maximálisan d mértékű indirekciót engedélyez, formálisan:

$$(f,g) \in EA^{(d)} \overset{\text{def}}{\iff} \exists\, d' \leq d : (f,g) \in EA^{(d')}_{call} \cup EA^{(d')}_{ret} \cup EA^{(d')}_{seq}.$$

Az *Execute Before* reláció ($EB^{(d)}$) a szimmetria alapján bármely d értékre egyszerűen a két metódus szerepének felcserélésével számolható:

$$(f,g) \in EB^{(d)} \overset{\text{def}}{\iff} (g,f) \in EA^{(d)},$$

és a fenti két reláció kombinálásával megkaphatjuk az *Execute Round* (ER) relációt is a következőképpen:

$$\forall d : ER^{(d)} = EB^{(d)} \cup EA^{(d)}.$$

Megfigyelhető, hogy a mi definíciónk speciális esete, az $EA^{(\infty)}$ megfelel az Apiwattanapong és társai által megadott *Execute After* reláció definíciójának, míg $ER^{(\infty)}$ a

végrehajtott metódusok közötti teljes relációt (teljes gráfot) jelöli. Szintén egyszerűen ellenőrizhető, hogy az *Execute Round* reláció szimmetrikus, míg a másik kettő nem. Az is látható, hogy az *ER* kiszámításához az EA_{call} és EA_{ret} relációk közül (ezek szimmetriája miatt) szintén csak az egyikre van szükség. Ennek ellenére a teljesség kedvéért a fenti definíciókat használjuk.

Természetesen, ha egy d érték elegendő két metódus *Execute Round* relációval való összekapcsolásához, akkor ezt a két függvényt az *ER* minden ennél magasabb d' érték esetén is összekapcsolja. Így érdekes lehet minden f, g metóduspárra meghatározni azt a legalacsonyabb d értéket, amelyre a két metódus *ER* relációban van, ez pedig nem lesz más, mint a korábban informálisan már definiált *Dynamic Function Coupling* (DFC) metrika:

$$DFC(f,g) = \begin{cases} \min\{d \mid (f,g) \in ER^{(d)}\} & \text{ha létezik ilyen } d, \\ \infty & \text{különben.} \end{cases}$$

Látható, hogy $DFC(f,g) = DFC(g,f)$ és $DFC(f,f) = 0$ igaz lesz bármely két f és g metódusra.[2]

A fenti definíciók a program egy konkrét futására adottak, de könnyedén bővíthetők úgy, hogy több futást is figyelembe vegyenek. Nevezetesen, az *EA*, *EB* és *ER* relációkat a program többszöri végrehajtására megkaphatjuk az egyenkénti végrehajtásra vonatkozó relációk uniójaként, míg a *DFC* metrika bármely két függvényre az egyedi futtatásokra számolt metrikaértékek minimuma lesz.

A továbbiakban formálisan is megadjuk a hatáshalmazok fenti relációkon alapuló számítási módszerét. Egy programra, tesztesetek egy halmazára, és egy rögzített d értékre, a változott metódusok C halmazával számolt dinamikus változás hatáshalmaz a következő:

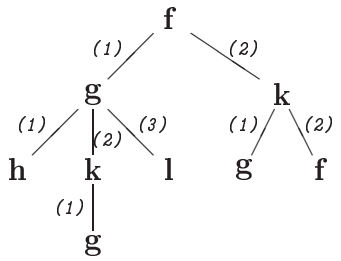
$$ImpactSet^{(d)}(C) = \{g \mid \exists f \in C : (f,g) \in ER^{(d)}\},$$

és hasonlóképpen, a számítási hatáshalmaz:

$$CImpactSet^{(d)}(C) = \{g \mid \exists f \in C : (f,g) \in EA^{(d)}\}.$$

Példa. Legyen $T = \langle f_e, g_e, h_e, h_r, k_e, g_e, g_r, k_r, l_e, l_r, g_r, k_e, g_e, g_r, f_e, f_r, k_r, f_r \rangle$ a bevezetőben példaként szereplő végrehajtási út, ez a 4.1 ábrán látható dinamikus hívási fát generálja.

[2] A tradicionális mértékekkel ellentétben itt a kisebb érték jelenti az erősebb kapcsolatot.



Az egy pontból induló élek számozása a hívási sorrendet jelenti.

4.1. ábra. Példa: dinamikus hívási fa

A 4.1 táblázat bal oldalán láthatók az EA_{call}-hoz, illetve jobb oldalán az EA_{seq}-hoz tartozó minimális d értékek ($\min\{d \mid (sor, oszlop) \in EA_{call}^{(d)}\}$ és $\min\{d \mid (sor, oszlop) \in EA_{seq}^{(d)}\}$).

d	f	g	h	k	l
f	0	1	2	1	2
g	∞	0	1	1	1
h	∞	∞	0	∞	∞
k	1	1	∞	0	∞
l	∞	∞	∞	∞	0

EA_{call}

d	f	g	h	k	l
f	∞	∞	∞	∞	∞
g	1	2	∞	1	2
h	3	2	∞	1	1
k	3	1	∞	2	1
l	3	3	∞	2	∞

EA_{seq}

4.1. táblázat. Példa: minimális d értékek az EA két komponenséhez

A két mátrix transzponálásával és a négy mátrix kombinálásával (azaz adott cellára a négy mátrix megfelelő cellájában található értékek minimumát véve) a DFC metrikára a 4.2. táblázatban látható értékeket kapjuk.

DFC	f	g	h	k	l
f	0	1	2	1	2
g	1	0	1	1	1
h	2	1	0	1	1
k	1	1	1	0	1
l	2	1	1	1	0

4.2. táblázat. Példa: DFC értékek

4.3. Algoritmusok

A következőkben az Execute Round relációt kiszámító algoritmusokat adjuk meg, de ezek könnyedén módosíthatók úgy, hogy csak az Execute After relációt számolják. A 4.3.1. alfejezetben egy globális, a 4.3.2. alfejezetben egy igényvezérelt, a 4.3.3. alfejezetben pedig egy speciális igényvezérelt algoritmust adunk meg.

4.3.1. Globális algoritmus

Ebben az alfejezetben megadunk egy olyan algoritmust, amely minden egyes lehetséges függvénypárra kiszámolja a DFC metrika értékét. Bár az algoritmus által meghatározhatóak az $ER^{(d)}$ relációk és a hatáshalmazok is, az algoritmus komplexitása miatt erre inkább a következő fejezetben bemutatott igényvezérelt algoritmust ajánljuk.

Az algoritmus egy T-vel jelölt, t hosszúságú végrehajtási úton dolgozik, ami lényegében metódus belépési és kilépési (entry/return) események sorozata. A T alapján épített dinamikus hívási fa maximális mélységét m-mel, míg az érintett metódusok darabszámát n-nel jelöljük.

A 4.2. ábrán látható algoritmus lelke a 4.3. ábrán látható rekurzív ComputeDistances függvény, ami a dinamikus hívási fa egy h metódusából induló aktuális részfájára a következőképpen működik. Először h minden egyes részfájára (7. sor) rekurzívan kiszámolja a minimális $h \to f$ hívási lánc hosszakat (8. sor), majd bármely két f és g függvény között a h-n keresztül ($h \to f$ és $h \to g$ hívási láncok által) megvalósuló szekvenciális indirekció minimális szintjét (9–13. sorok). Ezek után frissíti a DFC metrikát az ebben a h gyökerű részfában számolt h-ból induló hívási indirekció szintekkel (16–19. sorok), amikkel azután vissza is tér (miután eggyel növelte értéküket, hogy az éppen feldolgozott h gyökerű részfát tartalmazó részfában helyes eredményt kapjunk). Az algoritmus időigénye $O(t \cdot n^2)$, míg tárigénye a DFC metrika tárolásához szükséges hely nélkül $O(n \cdot m)$.

4.3.2. Igényvezérelt algoritmus

Az előző fejezetben ismertetett algoritmus globális abban az értelemben, hogy az összes lehetséges függvénypárra kiszámolja a DFC metrika értékét. A kiszámolt értékek alapján egyszerűen számolható a megváltozott függvények egy adott halmazához tartozó hatáshalmaz. A gyakorlatban viszont általában csak néhány függvény hatáshalmazára vagyunk kíváncsiak, így a globális algoritmus nem hatékony.

A 4.4., 4.5. és 4.6 ábrákon látható algoritmus változott függvények egy adott halmazához számítja ki az ER reláció és egy adott d indirekciós szint alapján a hatáshalmazt.

```
      program ComputeDFC(T)
      input: T : végrehajtási nyom
             F : összes függvények halmaza
      output: D[f,g] : DFC értéke az összes (f,g) függvénypárra
      begin
 1:       inicializáljuk D összes elemét ∞-re
 2:       E := Read(T)
 3:       ComputeDistances(E.function)
 4:       Output(D)
      end
```

4.2. ábra. ComputeDFC algoritmus: főprogram

```
      procedure ComputeDistances(h)
      input: h : függvény
      global: D[.,.], T, F
      local: P[f] : a h-tól mért előző távolság az összes f függvényre
             N[f] : a h-tól mért következő távolság az összes f függvényre
      begin
 5:       inicializáljuk P összes elemét ∞-re
 6:       E := Read(T)
 7:       while E egy ENTRY esemény do
 8:           N := ComputeDistances(E.function)
 9:           forall f ∈ F do
10:               forall g ∈ F do
11:                   D[f,g] := min(D[f,g], P[f] + N[g] − 1)
12:                   D[g,f] := D[f,g]
                  endforall
              endforall
13:           P := min(P,N)
14:           E := Read(T)
          endwhile
15:       P[h] := 0
16:       forall f ∈ F do
17:           D[h,f] := min(D[h,f], P[f])
18:           D[f,h] := D[h,f]
19:           P[f] := P[f] + 1
          endforall
20:       return P
      end
```

4.3. ábra. ComputeDFC algoritmus: távolságszámítás

A végrehajtási nyom feldolgozása során az algoritmus $n + 2$ darab vermet használ. A *CALL* és *RET* vermek tetején lévő értékek mutatják az utolsó változott függvénytől mért indirekciós szinteket a EA_{call} és EA_{ret} relációknak megfelelően, míg a $SEQ[g]$ teteje a

```
      program ComputeER(T, C, d)
      input: T : végrehajtási nyom
             F : összes függvények halmaza
             C : változott függvények halmaza
             d : indirekciós szint küszöbértéke
      output: IMP : a C változás hatáshalmaza
      data: CALL, RET : globális vermek call és ret indirekciókhoz
            SEQ[f] : vermek seq indirekcióhoz, minden f függvényre
      begin
1:        IMP := C
2:        inicializáljunk minden vermet egy ∞ berakásával
3:        while T ≠ ∅ do
4:            E := Read(T)
5:            if E egy ENTRY esemény then
6:                UpdateOnEntry(E.function)
              else
7:                UpdateOnReturn(E.function)
              endif
          endwhile
8:        Output(IMP)
      end
```

4.4. ábra. ComputeER algoritmus: főprogram

g-től számított EA_{seq}-nek megfelelő legkisebb távolságot tartalmazza.

Egy belépési (entry) esemény hatására (4.5 ábra) a vermek tetejére új értékek kerülnek. Ha az éppen feldolgozott f függvény eleme a változott függvények C halmazának, akkor a hívási indirekció értéke 0 lesz (mert ilyen távolságra vagyunk EA_{call} alapján egy változott függvénytől) különben eggyel nő (8–11. sor). A visszatérési indirekcióról hívás esetén még nem tudunk mit mondani (12. sor). A szekvenciális indirekció pedig minden függvényre eggyel nő (13–14. sor) akkor is, ha f egy változott függvény, mert kifejezetten az EA_{seq} alapján dolgozunk, aminek nem része az EA_{call}. Ezek után a hatáshalmazt frissítjük, ha valamely $d' < d$-re és g változott függvényre $(g, f) \in EA_{call}^{(d')}$ (15–16. sor), $(g, f) \in EA_{seq}^{(d')}$ (17–19. sor) vagy $(g, f) \in EB_{seq}^{(d')}$ (20–23. sor).

Egy kilépési (return) esemény hatására (4.6 ábra) minden veremből kivesszük a legfelső elemet (24., 25. és 30. sor), az alatta lévőt (ami a verem új teteje) pedig szükség esetén módosítjuk. A hívási veremben módosításra nincs szükség. A visszatérési indirekció új értéke 1 lesz ha változott függvényből lépünk vissza, illetve lehet eggyel nagyobb mint az éppen elhagyott függvényben, vagy maradhat változatlan ha f hívójának egy másik híváson keresztül már eleve egy ennél kisebb EA_{ret} szerinti indirekciós értéke van (25–28. sor). A visszatérési indirekcióhoz hasonlóan lehet frissíteni a szekvenciális indi-

```
      procedure UpdateOnEntry(f)
      input: f : függvény
      global: T, F, C, d, IMP, CALL, RET, SEQ[.]
      begin
 8:       if f ∈ C then
 9:           push(0, CALL))
10:       else
11:           push(top(CALL) + 1, CALL)
          endif
12:       push(RET, ∞)
13:       forall g ∈ F do
14:           push(SEQ[g], top(SEQ[g]) + 1)
          endforall
15:       if top(CALL) < d then
16:           IMP := IMP ∪ {f}
          endif
17:       forall g ∈ C do
18:           if top(SEQ[g]) < d then
19:               IMP := IMP ∪ {f}
              endif
          endforall
20:       if f ∈ C then
21:           forall g ∈ F − IMP do
22:               if top(SEQ[g]) < d then
23:                   IMP := IMP ∪ {g}
                  endif
              endforall
          endif
      end
```

4.5. ábra. ComputeER algoritmus: belépési esemény

rekciót tároló halmazokat is azzal a különbséggel, hogy ha változott függvényből lépünk vissza, akkor az indirekció új értéke a 0 technikai érték lesz (29–33. sor). Ezek után frissítjük a hatáshalmazt, ha valamely $d' < d$-re és g változott függvényre $(g, f) \in EA_{ret}^{(d')}$ (34–35. sor).

Az algoritmus minden T-beli elemet feldolgoz, és egy-egy ilyen elem feldolgozásánál a függvények számától lineárisan függő mennyiségű műveletet végez. A veremműveletek konstans időben elvégezhetőek, a halmazok pedig tárolhatóak oly módon, hogy a halmazműveletek szintén konstans idejűek legyenek (és ez a végső tárigényt lényegében ne befolyásolja). A szükséges tárhely a vermek számával és maximális méretével arányos. Időigénye tehát megfelelő adatstruktúrák használata esetén $O(t \cdot n)$, a tárigénye $O(n \cdot m)$.

```
      procedure UpdateOnReturn(f)
      input: f : függvény
      global: T, F, C, d, IMP, CALL, RET, SEQ[.]
      begin
24:       pop(CALL)
25:       u := min(pop(RET) + 1,pop(RET))
26:       if f ∈ C then
27:           push(RET, 1)
          else
28:           push(RET, u)
          endif
29:       forall g do
30:           u := min(pop(SEQ[g]) + 1,pop(SEQ[g]))
31:           if f = g then
32:               push(SEQ[g], 0)
              else
33:               push(SEQ[g], u)
              endif
          endforall
34:       if top(RET) < d then
35:           IMP := IMP ∪ {f}
          endif
      end
```

4.6. ábra. ComputeER algoritmus: visszatérési esemény

4.3.3. Igényvezérelt algoritmus *d=1*-re

Az előző fejezetben bemutatott algoritmust tovább egyszerűsíthetjük, ha az indirekció mértékére is mondunk egy korlátot. Nevezetesen, az előző algoritmus $d = 1$-re specializált változatát a 4.7., a 4.8. és a 4.9. ábrákon láthatjuk[3], ami az $ER^{(1)}$ reláció alapján számolja ki a hatáshalmazt.

Az algoritmus az általános algoritmushoz nagyon hasonlóan működik, de a használt adatszerkezetek egyszerűsödnek. A *CALL* verem teteje itt egy egyszerű logikai érték, ami azt mutatja, hogy a hívó függvény változott függvény-e. A *RET* verem teteje akkor igaz, ha legalább egy változott függvényből már visszatértünk az aktuális függvénybe. A *SEQ* adatszerkezetre az $EB^{(1)}_{seq}$ miatt van szükség, ez fogja a hívási fa egy pontjához tartozó korábbi testvéreket tárolni. A *SEQ* most egyetlen verem, ami azon függvények halmazait tárolja, amelyek visszatérnek az aktuális függvénybe. Az algoritmus (4.7. ábra) kezdetben inicializálja ezeket a vermeket (2–4. sor), majd elkezdi feldolgozni a T végrehajtási

[3]A firewall szó a névben a „Firewall Testing” kifejezésből adódik, ami a tesztelendő elemek közé a megváltozottakon felül az azokhoz valamilyen hatásanalízis szerint közvetlenül kapcsolódó elemeket veszi be. A $d = 1$ érték éppen a közvetlen kapcsolatokat jelenti.

```
      program ComputeFirewallER(T, C)
      input: T : végrehajtási nyom
             F : összes függvények halmaza
             C : változott függvények halmaza
      output: IMP : a C változás hatáshalmaza
      data: CALL, RET : globális vermek call és ret indirekciókhoz
            SEQ : globális verem seq indirekcióhoz
      begin
 1:       IMP := C
 2:       push(CALL, FALSE)
 3:       push(RET, FALSE)
 4:       push(SEQ, ∅)
 5:       while T ≠ ∅ do
 6:           E := Read(T)
 7:           if E egy ENTRY esemény then
 8:               UpdateFirewallOnEntry(E.function)
              else
 9:               UpdateFirewallOnReturn(E.function)
              endif
          endwhile
10:       Output(IMP)
      end
```

4.7. ábra. ComputeFirewallER algoritmus: főprogram

nyom eseményeit (5–9. sor).

Egy belépési (entry) esemény hatására (4.8 ábra) először az aktuális függvényt a *call* és *seq* részrelációk alapján felvesszük a hatáshalmazba, vagy éppen kihagyjuk belőle (10–17. sor). Az *IMP* halmazba kerüléshez szükséges információ a *CALL* és *RET* vermek tetején (a módosításuk előtt) direktben elérhető (10. és 12. sor), és a *SEQ* halmaz tetején lévő halmazból kinyerhető (15. sor). Ezután az alapján, hogy f változott függvény-e frissítjük a *CALL* vermet (18. sor). A másik két verembe `FALSE` illetve $\emptyset$ kerül (19. és 20. sor), mivel a bennük tárolt f-re vonatkozó információk majd egy f-be visszatérés során lesznek értelmezhetőek.

Egy kilépési (return) esemény hatására (4.9 ábra) a *ret* részreláció alapján lehet egy függvényt a hatáshalmazba felvenni (21–22. sor). Azt, hogy bele kell-e venni, közvetlenül kiolvashatjuk a *RET* verem tetejéről (21. sor). Ezután a vermekből egyszerűen kiürítjük a legfelső elemeket (23–25. sor), majd a *RET* és *SEQ* vermek új legfelső elemeit frissítjük (26–29. sor). Ha változott függvényből tértünk vissza, akkor a *RET* tetejét igazra állítjuk, különben marad, ami volt (26–27. sor). A *SEQ* legfelső halmazát pedig bővíthetjük az f függvénnyel, de mivel a halmazt csak arra használjuk, hogy a benne lévő elemekkel esetleg bővítsük az hatáshalmazt, a hatáshalmazban már benne lévő elemeket

```
      program UpdateFirewallOnEntry(h)
      input: h : függvény
      global: C, F, IMP, CALL, RET, SEQ
      begin
10:       if top(CALL) then
11:           IMP := IMP ∪ {f}
          endif
12:       if top(RET) then
13:           IMP := IMP ∪ {f}
          endif
14:       if f ∈ C then
15:           forall g ∈ top(SEQ) do
16:               IMP := IMP ∪ {g}
17:               top(SEQ) := top(SEQ) − {g}
              endforall
          endif
18:       push(CALL, f ∈ C)
19:       push(RET, FALSE)
20:       push(SEQ, ∅)
      end
```

4.8. ábra. ComputeFirewallER algoritmus: belépési esemény

```
      program UpdateFirewallOnReturn(h)
      input: h : függvény
      global: C, F, IMP, CALL, RET, SEQ
      begin
21:       if top(RET) then
22:           IMP := IMP ∪ {f}
          endif
23:       pop(CALL)
24:       pop(RET)
25:       pop(SEQ)
26:       if f ∈ C then
27:           top(RET) := TRUE
          endif
28:       if f ∉ IMP then
29:           top(SEQ) := top(SEQ) ∪ {f}
          endif
      end
```

4.9. ábra. ComputeFirewallER algoritmus: visszatérési esemény

felesleges beletenni (28–29.sor).

A hasonló felépítés miatt a legrosszabb művelet- és tárigény $O(t \cdot n)$ és $O(n \cdot m)$, mint az előző esetben (bár a két algoritmus között így is egy jelentős konstans szorzóbeli eltérés

van). Viszont míg átlagos esetben a ComputeER algoritmus nem igazán javul, addig a ComputeFirewallER összességében majdnem $O(t)$ idejűvé tehető. Ehhez csak a *SEQ* halmazainak implementációját kell úgy megválasztani, hogy az elemek felsorolása ne fixen $O(n)$ legyen, hanem a halmaz tényleges méretétől függjön; illetve azt kell felismerni, hogy a 17. és 28. sorok miatt a 15. sor **forall** ciklusának az összes lefutása összesen maximum $O(n \cdot m)$, de átlagosan csak $O(n + m)$ idejű.

4.4. Mérések és kiértékelés

A *DFC* metrika használhatóságával kapcsolatban több konkrét kérdést is megfogalmaztunk, amelyeket azután mérések segítségével sikerült megválaszolni. A kérdések megfogalmazásánál a következő szempontokat vettük figyelembe: (1) a hatáshalmazok méretéből általában következtethetünk használhatóságukra, azaz minél kisebb egy halmaz, annál nagyobb segítséget jelent a fejlesztők/tesztelők számára; (2) másrészt viszont nem mindegy a halmaz minősége, amit a *pontosság* és *teljesség*[4] kettőssel fejezhetünk ki. A kérdések a következők:

1. Kisebb DFC érték valóban valószínűbb kapcsolatot jelent?
2. A hívási (és visszatérési) rész-reláció mennyire fedi a függvények közötti valódi kapcsolatokat?
3. Mi az a d érték, aminél a relációk teljessége már megfelelő, és mennyi ekkor a pontosságuk?
4. Milyen d értéket használjunk, ha a pontosság a fontos, és ilyenkor milyen a teljesség?
5. Az eredeti EA relációhoz képest mekkora hatáshalmaz méret csökkenéssel számolhatunk?

Ebben a fejezetben bemutatjuk a mérési környezetet, majd a fenti kérdések megválaszolására végzett méréseink eredményeit.

4.4.1. Mérési környezet

A méréseinket három kisméretű nyílt forráskódú Java programon és a hozzájuk tartozó teszteseteken végeztük. A három program a *JSubtitles* 15 osztállyal és 460 kódsor-

[4] A teljesség és a biztonságosság majdnem ugyanaz a fogalom, egy 100%-os teljességű halmaz biztonságos, egy kevésbé teljes nem az.

ral, a *NanoXML* 27 osztállyal és 1156 sorral, valamint a *java2html* 55 osztállyal és 2290 sorral. Mindegyikhez 95–100 különböző tesztesetet használtunk.

A *pontosság* és *teljesség* meghatározásához pontos relációként utasítás szintű dinamikus szeleteket vettünk alapul, amiket a *Jadys* nevű Java dinamikus szeletelőnkkel [55] számoltunk. (A rendszer leírása megtalálható a dolgozat 7.3. fejezetében.) Az így kapott utasításszintű szeleteket azután metódusok szintjére emeltük, és ezeket a metódusok közötti kapcsolatokat használtuk mint valódi függőségeket.

A mérések során nem csak a *DFC* metrikával (illetve az általa meghatározott hatáshalmazzal), hanem a részrelációival és azok különféle kombinációival is foglalkoztunk. Természetesen az egyes relációkat mindig a nekik megfelelő irányú szeletekhez hasonlítottuk.

Meg kell említenünk, hogy a mérések során a teszteseteket nem külön-külön kezeltük, hanem a különböző tesztesetekre előállított eredményeket unióztuk. Mivel azonban az *ER* relációt (aminek a finomításaként kaptuk a *DFC* metrikát) a statikus relációk közelítő relációjának szánták (és eleve fel is készítették a több futtatásból származó dinamikus információk kezelésére), ezt a döntésünket nem problémának kell tekinteni, hanem sokkal inkább a mérések egy adottságának, és az eredményeket ennek tükrében kell értelmezni. A mérésekből levont konklúziók érvényességét ez nem befolyásolja.

4.4.2. Pontosság és teljesség

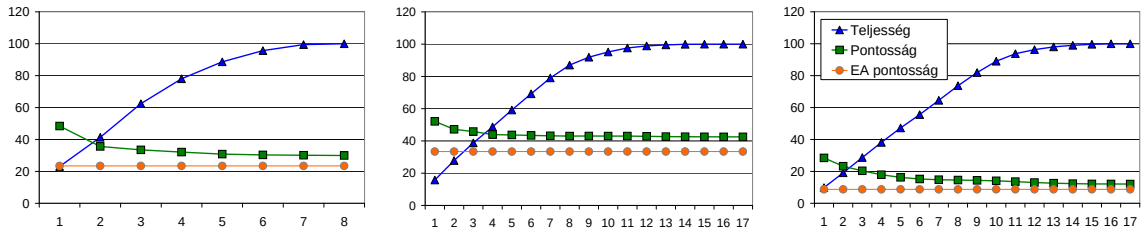
A mi esetünkben a *pontosság* azt a mérőszámot jelenti, ami megmutatja, hogy az adott algoritmus által kiszámolt hatáshalmaznak mekkora részét fedi le a valódi hatáshalmaz. Ez tehát egy 0 és 1 közötti szám, az algoritmus által visszaadott valódi függőségek és az algoritmus által visszaadott összes függőség számának hányadosa. Ha ez az érték 1, akkor az algoritmus csak valódi függőségeket fog visszaadni, és minél kisebb, annál több elemmel dolgozunk feleslegesen a hatáshalmaz használata során. A *teljesség* pedig azt mutatja, hogy a valódi hatáshalmaznak mekkora részét fedi le az adott algoritmus által kiszámolt hatáshalmaz. Ez is egy $[0, 1]$ intervallumba eső érték, az algoritmus által visszaadott valódi függőségek és az összes valódi függőség hányadosa. Ha ez az érték 1, akkor az algoritmus biztonságos, tehát egyetlen valódi függőséget sem mulasztunk el, és minél kisebb, annál kevésbé lehetünk biztosak a hatáshalmaz segítségével elért eredményekben.

Az optimális nyilvánvalóan egy 100% pontosságú, 100% teljességű algoritmus lenne, de általában a két érték fordítottan arányos. Azaz minél pontosabb az eredmény, annál kevésbé teljes (triviális példa a változott függvények halmaza, amelynek minden eleme nyilvánvalóan megvizsgálandó); illetve minél teljesebb, annál kevésbé pontos (a legtelje-

sebb ilyen halmaz például nyilvánvalóan az összes függvények halmaza). Hogy mikor kell inkább a pontosságra, és mikor inkább a teljességre törekedni, azt az aktuális alkalmazási szituáció dönti el. A legtöbb algoritmusban viszont nincs lehetőség a pontosság és teljesség egymáshoz viszonyított súlyának megváltoztatására. A mi algoritmusainkban viszont a d paraméter értékének módosításával bizonyos mértékig változtatható e két jellemző aránya.

A fentiek szellemében elvégeztünk egy sor pontosság/teljesség mérést. A részrelációk kombinációi közül három olyat találtunk, amelyet érdemes lehet használni: $EA_{call} \cup EA_{ret}$ ami tulajdonképpen az egy hívási láncon elhelyezkedő elemeket csoportosítja, EA ami az eredeti *Execute After* reláció és ER ami tulajdonképpen a DFC maga. Az első, csak hívási és visszatérési indirekciót tartalmazó reláció hasznos lehet ugyan az alacsony számítási igényei miatt ($O(t)$ időben megoldható), de a teljessége mindössze 22–28% ha $d = \infty$.

A 4.10. ábrán az ER reláció eredményei láthatóak. Mint látható, bár a konkrét adatpont-értékek eltérnek, a három ábrán látható görbék nagyon hasonlítanak egymáshoz. Megfigyelhető, hogy d növekedésével a teljesség is gyorsan – az elején nagyjából lineárisan – emelkedik, hogy valahol elérje a 100%-ot. Ez a d érték viszont programfüggő. Ebből arra következtetünk, hogy minden programhoz létezik egy olyan d érték, amivel az algoritmus már biztonságosan használható.



(a) JSubtitles (b) NanoXML (c) java2html

X *tengely: d értéke;* **Y** *tengely: teljesség és pontosság.*

4.10. ábra. Az ER reláció pontossága és teljessége

Ami a pontosságot illeti: ez mindhárom esetben eltérő értékről indul, de a d növekedésével nagyon gyorsan lecsökken egy viszonylag konstans, programfüggő értékre, ami az eredeti *Execute After* reláció pontossága felett helyezkedik el. A pontosság ennyire gyors csökkenése némileg meglepő volt számunkra.

4.4.3. Hatáshalmazok méretei

Természetesen a pontosság és a teljesség is fontos, de amit a felhasználó igazán megérez a módszer használata közben, az a halmazok mérete. A halmazok méreteit minden különböző d értékre kiszámítottuk, de mivel az ábrázolt görbék nagyon hasonlítanak a teljességi görbékre, az *ER* relációnak csak a $d = 1, 2$ eseteit mutatjuk be a 4.3. táblázatban. Az említett hasonlóság a $\frac{teljesség}{pontosság} \cdot C$ képlettel jellemezhető, melyben C egy konstans érték. Mivel a pontosság görbéin látható, hogy egy idő után majdnem konstanssá válnak, ebből következik, hogy a hatáshalmaz mérete a teljességgel együtt nő egy bizonyos értékig, amit 100%-os teljesség esetén ér el (bár a pontosság $d \to \infty$ esetén minimális mértékben tovább csökkenhet). Sajnos 100% teljességnél a hatáshalmazok méretei nem lényegesen kisebbek, mint az eredeti konzervatív EA módszerrel előállított hatáshalmazok.

	$d = 1$			$d = 2$		
	halmaz-méret	orig. p. pontosság	*ER* pontosság	halmaz-méret	orig. p. pontosság	*ER* pontosság
JSubtitles	14.3%	23.4%	48.4%	34.5%	23.4%	35.6%
NanoXML	13.5%	33.4%	52.1%	25.5%	33.4%	47.2%
java2html	4.2%	8.7%	28.5%	9.9%	8.7%	23.2%

4.3. táblázat. Hatáshalmaz méretek ER relációhoz

A 4.3. táblázatban az összes függvény hatáshalmazainak átlagos méretét és pontosságát mutatjuk. Látható, hogy $d = 1$-re az *ER* pontossága legalább kétszerese az eredeti *Execute After* algoritmus pontosságának, és még $d = 2$ esetben is nagyobb annál. Ugyanakkor a keletkezett hatáshalmazok jóval kisebbek, bár a teljességük messze elmarad az eredeti algoritmus 100%-os értékétől. Mindenesetre ez azt mutatja, hogy az $ER^{(d)}$ reláció sokkal jobb választás, mint az eredeti algoritmus által produkált hatáshalmazból véletlenszerűen kiválasztani ugyanennyi elemet.

4.4.4. Kiértékelés

Összefoglalva tehát a mérési eredményeket, a feltett kérdéseinkre a következő válaszokat adhatjuk.

1. **Kisebb DFC érték valóban valószínűbb kapcsolatot jelent?** Igen. A méréseink szerint a reláció pontossága a d érték növelésével nagyon meredeken és 1-2 lépésen belül egy onnantól kezdve nagyjából konstans értékre csökken. Tehát alacsony d esetén nagyobb a valószínűsége, hogy a megtalált kapcsolat valódi.

2. **A hívási (és visszatérési) rész-reláció mennyire fedi a függvények közötti valódi kapcsolatokat?** Kevéssé. A hívási és szekvenciális részrelációk hozzájárulása magas d érték esetén nagyjából 1 : 4 arányban stabilizálódik.

3. **Mi az a d érték, aminél a relációk teljessége már megfelelő, és mennyi ekkor a pontosságuk?** A mérések alapján ez ugyan programfüggő, de a teljesség viszonylag hamar (5–15 lépés a méréseink során) és nagyjából egyenletesen fut fel 100% közelébe. A pontosság viszont sokkal drasztikusabban csökken, gyakorlatilag 1-2 lépésen belül megközelíti (felülről) az eredeti EA reláció pontosságát.

4. **Milyen d értéket használjunk, ha a pontosság a fontos, és ilyenkor milyen a teljesség?** Méréseink alapján egyértelműen 1 vagy 2. A teljesség viszont ekkor még nagyon alacsony, 20% körül mozog.

5. **Az eredeti EA relációhoz képest mekkora hatáshalmaz méret csökkenéssel számolhatunk?** Alacsony d érték esetén jelentős, 65–95% közötti csökkenéssel, nagyobb d értékek esetén viszont a csökkenés (programfüggő mértékben, de nagyjából lineárisan) erőteljesen redukálódik.

4.5. Összefoglalás

A fejezetben definiáltuk a függvények között értelmezett *Dynamic Function Coupling* (DFC) metrikát, valamint megadtuk az őt, illetve az általa definiált hatáshalmazokat kiszámító algoritmusokat. Bemutattuk továbbá a metrikával számolt hatáshalmazokon végzett méréseinket, melyek alapján mondhatjuk, hogy a DFC által képviselt csatolásnak van alapja, vagyis az általa közelinek ítélt függvények között nagyobb valószínűséggel van kapcsolat, mint a távolinak ítéltek között. Meghatároztuk továbbá a metrikához azt az indirekciós értéket, amivel számolni érdemes ha az EA-val szembeni pontosságát szeretnénk kihasználni; ez nagyon alacsony értéknek adódott (1 vagy 2).

II. rész

Alacsony szintű analízis

5. fejezet

Programszeletelés

Az előző részben megismerkedhettünk a hatásanalízissel, lehetséges alkalmazási területeivel, illetve különféle hatáshalmaz számítási módszerekkel. A hatásanalízist magas szintű programanalízisként kategorizáltuk, hiszen függvénynél/metódusnál alacsonyabb szintű programelemekkel nem foglalkozott. Ebben a részben egy másikfajta programanalízist, a programszeletelést vesszük elő. Ha nagyon leegyszerűsítjük, a kétféle megközelítés nem különbözik: mindkettőnek ugyanaz az alapja: programegységek közötti függőségek felderítése. De a hatásanalízissel szemben a *programszeletelés* egységei az utasítások, ezért alacsony szintűnek nevezzük. Ebből kifolyólag sokkal részletesebben ábrázolhatjuk a programunkban található összefüggéseket, ez viszont nyilvánvalóan megnövekedett számításigénnyel jár. Bármennyire is igaz tehát, hogy a programszeletelés bizonyos szempontból hatásanalízis alkalmazásokban is megállja a helyét, sőt, sokkal megbízhatóbb és használhatóbb (pontosabb és biztonságosabb) eredményeket produkál, a megnövekedett számításigény miatt a szeletekből számolt magasabb szintű hatáshalmazokat általában nem használjuk.

Mindazonáltal a programszeletelésnek is megvannak azok az alkalmazási területei, ahol a magas szintű hatásanalízis használhatatlan eredményeket produkálna. Használható a *program megértés* (program understanding) területén, amikor például olyan biztonságilag kritikus kódrészleteket kell megtalálni, amelyek átszövik a teljes rendszert [13]. A szeletelés segítségével megtalálhatóak például azok a kódrészletek, amelyekben egy biztonságilag kritikus komponenshez tartozó változót használnak. Egy másik terület a *visszatervezés* (reverse engineering), amikor a programkódból kell következtetni a program felépítésére azért, hogy megállapíthassuk, mennyire tér ez el az eredetileg tervezettől. A szeletelés felhasználható ehhez a tevékenységhez. Ha például a programot szeletek hálójával reprezentáljuk, ahol a háló rendezési relációja a „szelete” reláció, akkor ezen a hálón keresztül a programozó megtalálhatja azokat a helyeket, amelyek a

visszatervezés szempontjából fontosak a számára [13, 8]. Hasznos lehet még *regressziós tesztelésnél* (regression testing), ahol a hatásanalízis alkalmazásához hasonlóan a teszteset kiválasztáson keresztül a tesztelés költségét csökkentheti [4]. De a programszeletelés ezeken túl használható még, nyomkövetésre, karbantartásra és sok egyéb dologra is [24, 51, 25, 56].

Programszeletelésre nagyos sokféle módszer létezik, és a programszeletelés témaköréről több remek összefoglaló is készült különböző szerzőktől: Tip [56], De Lucia [20], Kamkar [36], Xu és társai [62]. Itt most meg sem kíséreljük a szeleteléssel kapcsolatos összes eddigi irodalom feldolgozását. Csupán annyit teszünk, hogy az általunk is kutatott részterületét egy kicsit jobban körbejárjuk, illetve kitekintést adunk egyéb részterületekre, hogy a dinamikus szeletelést el tudjuk helyezni valahol a programszeletelés nagy témakörén belül.

5.1. Szeletelés fogalma

A *Programszeletelés* eredeti definícióját Mark Weiser adta meg [59] azon utasítások összességeként, amik hatással lehetnek a program egy *szeletelési kritériummal* meghatározott viselkedésére. A szeletelési kritériumot egy programsor-változó párosként definiálta, az ehhez tartozó szelet pedig a program egy olyan futtatható részeként, aminek a viselkedése a kritérium tekintetében megegyezik a teljes program viselkedésével. Ezzel Weiser tulajdonképpen a hátramutató futtatható statikus szeleteket definiálta.

Azóta persze sokféle szelet definíció és szeletelési algoritmus született [56], és ezeket sokféleképpen csoportosíthatjuk.

Az egyik talán legfontosabb, az alkalmazhatóságot leginkább befolyásoló kategória a szeletek iránya. Beszélhetünk *előremutató (forward) és* hátramutató (backward) szeletekről. Előremutató szeletelés esetén arra vagyunk kíváncsiak, hogy egy adott p programpont (az abban kiszámított változóérték) mely később végrehajtott utasításokra lehet vagy van hatással, azaz mely utasításoknak befolyásolja a végrehajtását vagy a kiszámított értékét. Hátramutató szeletelés pedig azt szeretnénk tudni, hogy mely programpontok vannak (lehetnek) hatással egy adott p pontban kiszámított v változó értékére.[1] Előremutató szeleteket használhatunk például hatásanalízis alkalmazások esetén,

[1] A továbbiakban előfordulhat, hogy bár adott környezetben mind az előremutató, mind a hátramutató szeletelés használható, mi mégis csak a hátramutató szeletelés szóhasználatával élünk. Például a „kritériumra hatással van valami” és nem „a kritérium hatással van valamire”. Ez olyankor és azért van, amikor és mert az éppen ismertetett elmélet lényege az egyik irányból is látszik és a másik irányú szóhasználat csak felesleges (és sokadik alkalommal esetleg már bosszantó) szóismétlésnek tűnne. A hátramutató szóhasználatot pedig a hátramutató szeletelés elterjedtebb volta miatt választottuk.

hátramutató szeleteket hibakeresésnél vagy program megértés során.

Egy másikfajta csoportosítás lehet, hogy a szelet önmagában egy futtatható program, vagy csupán egy utasításhalmaz. Az első esetben megköveteljük, hogy a redukált program a kritériumban szereplő v változó és p programpont tekintetében az eredetivel azonos módon működjön, azaz a p ponton minden esetben ugyanazt az értéket számítsa ki v-be, mint az eredeti program. A második esetben a pedig „csak" annyi az előírás, hogy az összes olyan utasítást tartalmaznia kell, ami hatással van a szeletelési kritériumra.

Szintén fontos jellemzője a szeletelésnek, hogy egy szeletet a program egy konkrét lefutására szeretnénk meghatározni, vagy minden utasítást szeretnénk benne viszontlátni, aminek potenciálisan hatása lehet a kritériumra. Az első esetben nyilvánvaló, hogy a statikus elemzéssel elérhető információk mellett szükségünk van a futás során keletkező dinamikus információkra is, így ezt *dinamikus szeletelés*nek hívjuk. Ilyenkor a dinamikus szeletelési kritériumnak része lesz a program bemenete (hiszen hatással van a szeletre), és egy újabb elem ami azt mutatja, hogy az adott utasítás hányadik lefutása során vagyunk kíváncsiak a megfelelő változó értékére. A második esetben a szeleteket a kód végrehajtása nélkül, magából a statikus kódból nyerjük, ez a *statikus szeletelés*. A statikus szeletek nyilvánvalóan nagyobbak lesznek mint a dinamikus szeletek, hiszen egy adott kritériumhoz tartozó statikus szeletnek szükségszerűen tartalmaznia kell az összes ugyanahhoz a kritériumhoz tartozó dinamikus szeletet.

Az egyik – ha nem a – legalapvetőbb statikus szeletelési módszert Horwitz és társai publikálták [30]. A programot egy *rendszerfüggőségi gráf* (*System Dependence Graph* – SDG) segítségével ábrázolják, ami egy meglehetősen természetes leképezése a programnak és a programelemek között fellépő különböző (vezérlési-, adat-) függőségi relációknak (amik végül is meghatározzák a szeleteket). A gráfépítés után így a szeletelés már csak egy (többmenetes) gráfbejárás lesz. Több további statikus szeletelő algoritmusnak is ez a módszer az alapja.

A Korel és Laski [39, 40], Agrawal és társai [1, 3] illetve Kamkar és társai [37] által publikált alapvető dinamikus szeletelő algoritmusok többféle elvre épülnek. Korel például adatfolyam egyenleteket (dataflow equations) használ, és úgynevezett eltávolítható blokkokból (removable blocks) építi a szeleteket. Az egyik legelterjedtebb viszont az Agrawal és Horgan [3] által publikált *dinamikus függőségi gráf* (*Dynamic Dependence Graph* – DDG) alapú megközelítés. Ez tulajdonképpen az előbb említett SDG alapú statikus szeletelés dinamikus párja. A DDG minden egyes utasítás minden egyes előfordulásához (minden *akcióhoz*) egy külön csúcsot rendel, az élek pedig a program egy konkrét futása során keletkező dinamikus függőségeket reprezentálják. Maga a szeletelés – hasonlóan az SDG alapú statikus módszerhez – ebben az esetben is az előzőleg

felépített gráf egyszerű bejárása lesz.

Az *uniós szeletelés* alapgondolata, hogy sok, de véges számú dinamikus szelet uniójával közelíthető a statikus szelet [58]. Erre azért lehet szükség, mert statikus szeletelés közben általában eléggé konzervatív döntéseket hozunk olyan esetekben, amikor a dinamikus módszer egyértelműen megadja az eredményt. Például egy pointeren keresztüli olvasás potenciálisan bármilyen adatfüggőséget behozhat, míg dinamikusan egyértelmű, hogy honnan olvasunk. A kiszámolható statikus szeletek tehát általában messze vannak a *realizálható szelet*nek nevezett, elméletileg legkisebb statikus szelettől.

A *dekompozíciós szeletek* [25] az uniós szeletekhez hasonló kategóriát képviselnek olyan szempontból, hogy egy v változó bármely előfordulását befolyásoló utasításokat tartalmaznak, tehát tulajdonképpen az adott változó összes előfordulására számolt szeletek unióját kell meghatároznunk (a szeletelési kritérium egyetlen változó lesz). Hasznosak lehetnek például amikor karbantartás során változik a programkód, mert a változás által nem érintett programrészek meghatározhatóak általuk.

5.2. Dinamikus szeletelés

Számos alkalmazásban (például nyomkövetésnél) a dinamikus szeletek sokkal használhatóbbak, hiszen sokkal pontosabbak (praktikusan kisebbek) mint statikus társaik. Sajnos azonban a legtöbb alap algoritmusnak problémát okoz a valódi méretű inputok lekezelése. A dinamikus analízis ugyanis önmagában egy nehéz probléma. Ennek legfőbb oka, hogy a program futása során rengeteg esemény keletkezik, amit fel kell dolgozni. Ez rengeteg adatot jelent, különösképpen szeletelésnél, ahol a pontos eredmény elérése érdekében nagyon részletes számítások elvégzésére van szükség.

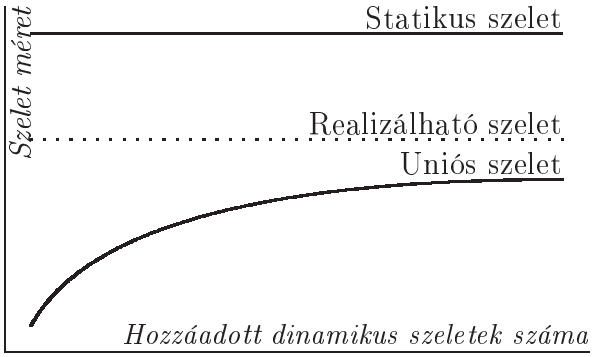
Az egyik legelterjedtebb dinamikus módszer a már említett DDG alapú szeletelés [3]. Nagy hátránya viszont, hogy a DDG mérete potenciálisan korlátlan, pontosabban a programban végrehajtott lépések (potenciálisan végtelen) számával egyenesen arányos. Bár maguk a szerzők is adtak javaslatot a DDG méretének csökkentésére, még ez a megoldás is túlságosan magas felső méretkorláttal ($O(2^n)$, ahol n a program utasításainak száma) rendelkezik [56] ahhoz, hogy a szeletelési módszert a gyakorlatban is használni lehessen. Zhang és Gupta is elkészítette a saját DDG tömörítő módszerét ami körülbelül egy nagyságrenddel kisebb gráfokat produkál [65], és más hatékony implementációkkal is foglalkoztak [66, 67]. Mund és Mall is adott egy statikus függőségi gráfokon alapuló megoldást, ahol a dinamikus információkat különféle plusz adatstruktúrákban tárolták [45].

Gyimóthy Tibor, Forgács Gábor és Beszédes Árpád elkészített egy olyan hátramutató szeleteket számító dinamikus algoritmust, amely a program futása során keletkezett

események egyszeri végigjárásával az összes lehetséges dinamikus kritériumra kiszámolja a megfelelő szeleteket [28]. Az algoritmus nem gráf alapú, hanem az egyes utasítások reprezentálására úgynevezett $d : U$ (*definíció-használat*) párokat használ, ami például egységesen kezeli az adat és vezérlési függőségeket. Bár az eredeti algoritmus csak egy nagyon egyszerű programozási nyelvet támogat, viszonylag alacsony tárigénye – ami nem a végrehajtott lépések, hanem a tárolt változók számával arányos – miatt alkalmas nagyméretű programok szeletelésére is. Ezt az algoritmust később valódi, C [BGS01, FG02] és Java [55] nyelvű programok szeletelésére is alkalmassá tettük. A dolgozat 6. és 7. fejezeteiben ennek az algoritmusnak az alapjaira építkező további algoritmusokat illetve ezek konkrét nyelvekre történő implementációinak részleteit mutatjuk be.

5.3. Uniós szeletek

Valódi programok esetén a statikus szeletek használhatatlanul nagyok lehetnek, köszönhetően annak, hogy futási információk nélkül pontos adat- és vezérlési függőséget nem, vagy csak nagyon nagy erőforrás-ráfordítással lehetne meghatározni. Problémát okozhatnak például a kései kötésű eljáráshívások, a mutatók vagy a tömbelemek elérése. Másrészt viszont dinamikus esetben mindezen információk a rendelkezésünkre állnak. Kézenfekvő tehát, hogy ha nagyon sok dinamikus szeletünk van ugyanarra a statikus kritériumra, akkor ezek uniójával a statikus szelet egy alsó közelítését kapjuk [58].



5.1. ábra. Realizálható szeletek közelítése

A statikus szelet viszont a dinamikus információk teljes hiánya és becsülhetetlensége miatt valószínűleg mindenképpen jóval nagyobb lesz, mint amekkora az összes lehetséges inputra a dinamikus szeletek uniója lenne. Ezt az elméleti legnagyobb uniós szeletet *realizálható szeletnek* hívjuk. A 5.1. ábrán a statikus, uniós és realizálható szeletek kapcsolatát szemléltetjük. Megjegyezzük, hogy amennyiben futtatható szeleteket követelünk meg, úgy a dinamikus szeletek uniója már nem feltétlenül marad valós szelet [21, 29].

Uniós szeletekkel mi is végeztünk méréseket, kísérleteket. A 8.2.2. fejezetben például uniós szeletek és a program lefedettség kapcsolatára vonatkozó mérési eredményeket is prezentálunk.

6. fejezet

d:U alapú programszeletelési algoritmusok

Gyimóthy Tibor és szerzőtársai kidolgoztak egy olyan dinamikus szeletelési algoritmust [28], amely a végrehajtás során keletkezett információk egyszeri feldolgozásával előállítja az összes dinamikus kritériumhoz a hátramutató szeleteket. Az algoritmus gráf reprezentáció helyett az egyes utasítások statikus függőségi információit úgynevezett *definíció-használat* (*definition-use*) adatpárokban tárolja, és így dinamikus függőségi gráfokat sem épít. Ugyanezt a reprezentációt használja a Beszédes Árpád által kidolgozott, egyetlen szeletelési kritériumhoz tartozó hátramutató szeletet kiszámító algoritmus is [9]. Ezek alapján nyilvánvalóvá vált, hogy ugyanezzel a reprezentációval többféle gráfmentes dinamikus szeletelő algoritmus is megalkotható. Ezért azután meghatároztuk a fentebb említett algoritmusok néhány jellemzőjét (szeletelés és feldolgozás iránya, egy menetben előállított szeletek száma), és megnéztük, hogy ezek a jellemzők milyen egyéb értékeket vehetnek fel, majd elemeztük az összes így előállt kombinációt.

Ebben a fejezetben tehát olyan $d : U$ alapú szeletelő algoritmusokat mutatunk be, amelyek nem igénylik előzetes dinamikus adatstruktúrák (mint például a DDG) felépítését, mégis ugyanazokat a szeleteket produkálják mint a sok helyen etalonként hivatkozott DDG alapú algoritmus. Az algoritmusok osztályozása, valamint a négy új algoritmus meghatározása és kidolgozása szerzőtársaimmal közös eredmény. A $d : U$ alapú algoritmusok és a DDG alapú algoritmus által számolt szeletek ekvivalenciájának bizonyításában saját munkám volt meghatározó.

6.1. Az algoritmusok osztályozása

Adott egy program és statikus $d : U$ reprezentációja és egy lefutásának *végrehajtási története* (execution history). Az ezeken a reprezentációkon dolgozó dinamikus szeletelő algoritmusokat a következő dimenziók szerint osztályozhatjuk.

Szeletek iránya. A két alapvető szeletelési irány az *előremutató* és a *hátramutató* szeletelés. Az előremutató szeletelés esetén arra vagyunk kíváncsiak, hogy a szeletelési kritérium által meghatározott értékek a program futása során mely későbbi programpontokon számolt értékek kiszámítására vannak hatással, azaz a kiszámított érték hol van (akár áttételesen is) felhasználva. A hátramutató szeletelésnél pedig arra kérdezünk rá, hogy az adott kritérium által meghatározott érték kiszámítására mely előző programpontokon számolt értékek vannak hatással, vagyis a kiszámított érték előállításához (akár áttételesen is) mit használtunk fel.

Globális vagy igényvezérelt. A tradicionális megközelítés szerint egyszerre egy kritériumunk van, és az ehhez tartozó egyetlen szeletet számítjuk ki. Ebben az esetben a szeletelés a kritériumban meghatározott programpontban indul, és a szeletelés irányának megfelelően haladunk tovább a különböző függőségek mentén valamelyik irányban. Ezt nevezzük *igényvezérelt* szeletelésnek. Lehetőség van viszont arra, hogy a végrehajtási történet egyszeri feldolgozásával egyszerre több (akár az összes lehetséges) kritériumhoz tartozó szeletet kiszámoljunk. Ebben az esetben *globális* szeletelésről beszélünk.

Feldolgozás iránya. A végrehajtási nyomot szintén kétféleképpen dolgozhatjuk fel. Az *előrefelé haladó* feldolgozás tekinthető „természetes” iránynak, hiszen a végrehajtási történet is ilyen módon keletkezik. Éppen ezért néha csak ez a fajta feldolgozási irány valósítható meg (például amikor a teljes nyom tárolása megvalósíthatatlan), párhuzamosítással ez az irány akkor is megoldható. Ugyanakkor előfordulnak olyan szituációk, amikor a *hátrafelé haladó* feldolgozás is alkalmazható és praktikusabb, mint a másik irány.

A fenti osztályozási dimenziók és értékeik alapján összesen 8 féle lehetséges algoritmus adódik, amiket a 6.1. táblázatban foglaltunk össze. A táblázat utolsó oszlopában az algoritmusok hasznosságára vonatkozó előzetes értékelés látható, a 6.2. fejezetben az algoritmusok bemutatásánál is ezt a csoportosítást használjuk. Nevezetesen, először (a 6.2.1. alfejezetben) a két igényvezérelt algoritmust mutatjuk be, amit (a 6.2.2. alfejezetben) a két praktikus globális majd (a 6.2.3. alfejezetben) a két párhuzamos globális

algoritmus követ. Ez összesen 6 algoritmus. A maradék két algoritmus megvalósítása értelmetlen, hiszen az igényvezérelt szeleteket fordított irányú feldolgozással[1] számolni gyakorlatilag ugyanannyi erőforrást igényel, mint egy ugyanilyen globális szeletelés, ami így nyilvánvalóan minden esetben alkalmazható az igényvezérelt helyett.

Globális/Igényvezérelt	Szeletelés iránya	Feldolgozás iránya	Hasznosság
Igényvezérelt	hátramutató	visszafelé	praktikus
Igényvezérelt	hátramutató	előrefelé	értelmetlen
Igényvezérelt	előremutató	visszafelé	értelmetlen
Igényvezérelt	előremutató	előrefelé	praktikus
Globális	hátramutató	visszafelé	párhuzamos
Globális	hátramutató	előrefelé	praktikus
Globális	előremutató	visszafelé	praktikus
Globális	előremutató	előrefelé	párhuzamos

6.1. táblázat. Dinamikus szeletelő algoritmusok áttekintése

6.2. Algoritmusok

Mielőtt belevágnánk az algoritmusok egyenkénti ismertetésébe, először bemutatjuk az általános működési elvüket, illetve bevezetünk néhány fogalmat és jelölést, amiket az algoritmusok részletes leírásánál használunk. Mivel ebben a fejezetben az alap algoritmusok ismertetése történik meg, egy egyszerű programnyelvre koncentrálunk. Így az itt bemutatott algoritmusok és adattípusok nem tartalmaznak semmilyen módosítást, ami egy valódi programnyelv szeleteléséhez szükséges volna.

A bemutatott algoritmusok mindegyike két adathalmazon dolgozik: a *végrehajtási történet*en (végrehajtási nyom, execution history) és a program egy tömör statikus reprezentációján. Előbbi egy egyszerű lista amelyben úgynevezett *akciók*, azaz utasítások futás közbeni előfordulásai vannak felsorolva olyan sorrendben, ahogy azok a program adott inputtal történő futtatása esetén követik egymást. Az algoritmusok ezt a nyomot dolgozzák fel előrefelé vagy visszafelé haladó irányban a dinamikus függőségek kinyerése céljából. Ehhez már csak egy *definíció-használat* (definition-use) relációt szükséges megadni a program minden utasítására, ami alapján az aktuális futás során valóban létrejövő dinamikus adatfüggőségi láncok kiszámolhatóak a végrehajtási nyom feldolgozása során. Ez azért elegendő, mert ha a program predikátumait (döntési utasításait,

[1] A fordított irány a szeletek és a feldolgozási irány viszonyára utal, tehát előremutató szeletek visszafelé haladó feldolgozással illetve hátramutató szeletek előrefelé haladó feldolgozással történő kiszámítását jelenti.

feltételeit) általános változókként kezeljük, akkor a vezérlési (kontroll) függőségeket is adatfüggőségként kezelhetjük.

A *végrehajtási történet* jelölése (az angol *Execution History* elnevezés nyomán) EH, ez i^j alakban jelölt *akció*k sorozatát tartalmazza, ahol i a végrehajtott utasítás programbeli sorszáma, míg j a végrehajtási lépés sorszáma a végrehajtási nyomban. Az adott programfutás során végrehajtott összes lépések számát J jelöli. Ezeken túl használni fogjuk a következő jelöléseket is:

$$i(i^j) = i,\ j(i^j) = j,\ EH = \langle {i_1}^1,\ {i_2}^2,\ \ldots,\ {i_J}^J \rangle\,.$$

Továbbá, $EHI(j) = i(i^j)$ jelöli a futtatás j-ik lépésében végrehajtott utasítás programbeli i sorszámát.

A szeletelési kritériumra a $CB = (\mathbf{x}, i^j, V)$ illetve $CF = (\mathbf{x}, i^j)$ jelöléseket fogjuk használni hátramutató (backward, CB) illetve előremutató (forward, CF) szeletelés esetén, ahol: $\mathbf{x}$ az az input, amivel a program futtatva lett; i^j az az akció, amelyre a dinamikus szeletet ki kell számolni; V pedig a program i-ik utasításában használt változók azon részhalmaza, amelyek szeleteire kíváncsiak vagyunk. Feltételezzük, hogy: (1) minden utasításban pontosan egy változót definiálunk, és nulla vagy több változót használunk; (2) hátramutató szeleteknél a V tartalmazza az összes, i-ik utasításban használt változót; (3) előremutató szeleteknél az i-ik utasításban definiált egyetlen változó szeleteit keressük. Ezek a feltételezések a használhatóság szempontjából semmilyen megkötést nem jelentenek, valóban csak az egyszerűség kedvéért ragaszkodunk hozzájuk. Előfordul, hogy a szövegben a CB és CF kritériumokhoz tartozó szeletek helyett egyszerűen csak az i^j akcióhoz tartozó szeletről beszélünk. Mivel az adott környezetben az input rögzített, valamint a fenti 3 pont alapján az akció a kritérium változóit is egyértelműen meghatározza, a kritérium és a megfelelő irányú szelet között bijektív megfeleltetés van, tehát az „akció szelete” szóhasználat helyénvaló rövidítés.

Az algoritmusok által használt statikus program reprezentációt a program $d : U$ reprezentációjának nevezzük. Lokális, vagyis az egy utasításon belül használt változók közötti definíció-használat kapcsolatokat tartalmaz a program minden egyes utasításához. Mivel feltételezzük, hogy minden utasítás pontosan egy változót definiál, a program egy utasításának $d : U$ reprezentációja:

$$i.\ d_i : U_i\,,$$

ahol i az utasítás sorszáma. A program összes utasításának számát I fogja jelölni. Az i-ik utasításban definiált változó $d(i) = d_i$, míg $U(i) = U_i$ jelöli a d_i kiszámításához ebben az utasításban felhasznált változók halmazát.

Ennek a megközelítésnek az egyik nagy előnye, hogy ugyanez a $d : U$ reprezentáció felhasználható nemcsak az adat, de a vezérlési függőségek ábrázolására is, ami jelentősen leegyszerűsíti a szeletelő algoritmusokat. Nevezetesen, bármely d_i definiált és $u_k \in U_i$ használt változó ($i = 1, \ldots, I$) lehet *predikátum változó*, speciális jelentéssel. Az ilyen predikátum változók virtuális változók, vagyis nem jelennek meg valódi változóként a programban, hanem a program minden egyes predikátumához (döntési utasítás, mint amilyen például az `if` vagy a `for`) generálunk egyet. A program döntési utasításai határozzák meg az utasítások közötti vezérlési függőségeket, tehát a megfelelő predikátum változókat definiált és használt változóként alkalmazva a vezérlési- és az adatfüggőségeket azonos módon kezelhetjük. Pontosabban, ha az i-ik utasítás egy döntési utasítás, akkor a definiált változója a p_i predikátum változó lesz (azaz $d(i) = p_i$), a felhasznált változói pedig az utasításban használt változók. Továbbá minden i' utasítás használt változóinak $U(i')$ halmazába beletesszük annak a $k_{i'}$ döntési utasításnak a $p_{k_{i'}}$ predikátum változóját, amivel i' direkt vezérlési függőségben áll (amitől függ).

A szeletelő algoritmusok formalizálása során használunk majd néhány további jelölést is. Egy v változó *utolsó definíciója* a j-ik végrehajtási lépés előtt az az $LD(v, j)$-vel jelölt akció, amelyre

$$LD(v, j) = i'^{j'}, \text{ ahol } j' < j \ \wedge \ d(i') = v \ \wedge \ \nexists j'' \, (j' < j'' < j, \ d(EHI(j'')) = v).$$

Továbbá az egyes algoritmusokban az i^j akciót feldolgozó lépésben szükség lehet a tetszőleges v változót *utoljára definiáló lépés*re, *utoljára definiáló utasítás*ra és *utoljára definiáló akció*ra, amikre rendre a rövidített $LD(v) = j(LD(v, j))$, $LS(v) = i(LD(v, j))$ és $LA(v) = LD(v, j)$ jelöléseket fogjuk használni. Nyilvánvaló, hogy előrehaladó feldolgozás esetén az i^j akció feldolgozása után $LD(d(i)) = j$, $LS(d(i)) = i$ és $LA(d(i)) = i^j$ mindaddig teljesül, amíg egy következő akció nem $d(i)$-t definiálja.

Az algoritmusok működését a 6.1. ábrán (a $d : U$ reprezentációval együtt) látható példaprogramon szemléltetjük. Ennek a kis példának érdekes végrehajtási történetei vannak az $\mathbf{x} = \langle \mathtt{a} = 0 \rangle$, $\mathbf{x} = \langle \mathtt{a} = 1 \rangle$ és $\mathbf{x} \in \{\langle \mathtt{a} = 2 \rangle, \langle \mathtt{a} = 3 \rangle, \ldots\}$ esetekben. A működés bemutatásához az $\mathbf{x} = \langle \mathtt{a} = 1 \rangle$ inputot, és az ehhez tartozó – a 6.2. ábrán is látható – $EH = \langle 1^1, 2^2, 3^3, 4^4, 5^5, 6^6, 7^7, 4^8, 8^9 \rangle$ végrehajtási nyomot használjuk.

6.2.1. Igényvezérelt algoritmusok

Az igényvezérelt algoritmusokat akkor használhatjuk, amikor egy konkrét programponthoz tartozó előre-, illetve hátramutató szeletekre vagyunk kíváncsiak. Ezek az algoritmusok egyszerűek olyan tekintetben, hogy csak ehhez az egyetlen kritériumhoz tartozó

i		$d : U$
1	`read(a)`	$a : \emptyset$
2	`y=0`	$y : \emptyset$
3	`x=1`	$x : \emptyset$
4	`while(a>0)`	$p : \{a\}$
5	`  y=x`	$y : \{x, p\}$
6	`  x=2`	$x : \{p\}$
7	`  a=a-1`	$a : \{a, p\}$
8	`z=y`	$z : \{y\}$

6.1. ábra. Példa: egy példaprogram és $d : U$ reprezentációja

i^j	statement	$d : U$
1^1	`read(a)`	$a :$
2^2	`y=0`	$y :$
3^3	`x=1`	$x :$
4^4	`while(a>0)`	$p : \{a\}$
5^5	`y=x`	$y : \{x, p\}$
6^6	`x=2`	$x : \{p\}$
7^7	`a=a-1`	$a : \{a, p\}$
4^8	`while(a>0)`	$p : \{a\}$
8^9	`z=y`	$z : \{y\}$

6.2. ábra. Példa: a példaprogram végrehajtási nyoma az $a = 1$ inputra

szelet kiszámítására koncentrálnak, így az ezen szelet kiszámításához nem használható lényegtelen információkat nem tárolják, azzal nem számolnak.

Összesen négyfajta igényvezérelt algoritmus lehetséges a kategorizálásunk szerint. De – amint azt az algoritmusok előzetes kategorizálásánál már jeleztük – a négy közül az a kettő, amelyben a szeletek és a feldolgozás iránya ellentétes (vagyis az előrefelé számolt hátramutató és a hátrafelé számolt előremutató szeletelés) gyakorlatilag értelmetlen. Ilyen esetben ugyanis egészen a kritériumban meghatározott akcióig minden akció szeleteit ki kellene számolni, bár később ezek közül jó néhány eldobható lenne. Tehát tulajdonképpen egy globális szeletelést kellene végrehajtani úgy, hogy a végén csak az egyik szeletet írjuk ki.

Így viszont két értelmes igényvezérelt algoritmus marad: a visszafelé számolt hátramutató és az előrefelé számolt előremutató. A két algoritmus hasonlóképpen működik. Mindkét esetben a szeletelési kritériumban meghatározott akciónál kezdjük a végrehajtási nyom feldolgozását. Azután a $d : U$ reprezentáció segítségével végigkövetjük a a dinamikus függéseket – a szeletelés irányától függően – egészen a végrehajtási nyom

legelső vagy legutolsó akciójáig.

Az előremutató szeletek számításánál az *EH* feldolgozása nem jelent gondot, lévén ez a „természetes” irány. Ugyanez a hátramutató szeleteknél már problémásabb, hiszen – mivel a nyomot visszafelé irányban dolgozzuk fel – szükség van az *EH* előzetes eltárolására (legalább a kritériumban meghatározott akcióig). Ráadásul így az utolsó definiáló utasítás kinyerése sem kézenfekvő. Emiatt a hátramutató szeleteket a nyom előrefelé haladó feldolgozásával kiszámító globális algoritmus sokkal inkább alkalmazhatónak tűnik akár egyetlen hátramutató szelet kiszámításához is.

Érdemes megemlíteni, hogy a feldolgozás iránya nem csak az igényvezérelt algoritmusoknál jelent egyfajta használhatósági osztályozást. Az összes algoritmus-párra igaz, hogy míg az előrefelé haladó algoritmusok a program futásával párhuzamosan képesek a szeletek előállítására (és így akár potenciálisan végtelen programok szeletelésére is használhatók), addig a hátrafelé haladó párjaik alkalmazásához először tárolni kell a teljes végrehajtási történetet, ami nem igazán nevezhető korlátos adatmennyiségnek.

Visszafelé számolt hátramutató

Ez az algoritmus Beszédes Árpád eredménye [9], de a teljesség kedvéért ezt is bemutatjuk. Az algoritmus működési elve a következő. Az algoritmus végigvizsgálja az akciók közötti dinamikus függőségeket, és a még fel nem dolgozott akciókat egy munkahalmazban (*worklist*) tárolja. Amikor kivesz egy elemet ebből a halmazból, megvizsgálja ennek az akciónak az összes használt változóját, és az ezen változókat az aktuális akció előtt utoljára definiáló akciókat felveszi a munkahalmazba. Amikor már minden függőséget megvizsgált és így a munkahalmaz is kiürült, az algoritmus a futása során érintett utasítások halmazát mint megfelelő szeletet visszaadva véget ér. A formalizált algoritmus a 6.3 ábrán látható.

Meg kell jegyeznünk, hogy az algoritmus 9. sorában található utoljára definiáló akció értékét a végrehajtási nyom még fel nem dolgozott részéből kell kiolvasnunk, és ez az információ ebben a részben sajnos bárhol lehet. Ezért az algoritmus hatékony működése érdekében a végrehajtási nyomot egy olyan speciális alakban kell eltárolni, amely a definiált változó szerint csoportosítja az akciókat. Ezt egy úgynevezett *EHT* (*Execution History Table*) tábla segítségével tehetjük meg, aminek egy-egy sora a megfelelő definiált változóhoz tartozó akciókat tartalmazza. Erre azért van szükség, mert az algoritmus minden iterációjában szükség lehet tetszőleges változókat definiáló akciókra, ami egy keresést jelent a végrehajtott akciók között. Így az $LA(u, l) = k'^{l'}$ értéke megkapható úgy, hogy először kiválasztjuk az *EHT*-ből az u-nak megfelelő sort, majd ebben megkeressük a megfelelő $l' < l$ értéket (ez már hatékonyan implementálható).

```
     program Algorithm-DB(P, CB)
     input: P : a program
            CB = (x, i^j, V) : a dinamikus szeletelési kritérium
                 (feltételezzük, hogy V = U(i))
     output: S : a P program CB kritériumhoz tartozó szelete
     begin
 1:      S := ∅
 2:      ReadStore(EH), az i^j akcióig
 3:      worklist ← i^j
 4:      while worklist ≠ ∅ do
 5:          k^l := a worklist halmazból vegyük ki a legmagasabb
                 l lépésszámmal rendelkező akciót
 6:          if l ≠ j then
 7:              S := S ∪ {k}
             endif
 8:          for ∀u ∈ U(k) do
 9:              worklist ← LD(u, l)
             endfor
         endwhile
10:      Output(S): a CB kritériumhoz tartozó szelet
     end
```

6.3. ábra. Algorithm-DB: igényvezérelt algoritmus hátramutató szeletekhez

Szintén megjegyzendő, hogy az algoritmus 5. sorában a „legmagasabb l lépésszámmal rendelkező” kitétel nem szükséges, de hasznos lehet az *EHT* hatékony implementálásához. Nevezetesen, így az *EHT* egy adott sorából a megtalált elem mögötti összes akció törölhető lesz.

Az algoritmus működését a 6.1. ábrán látható program $CB = (\langle a = 1\rangle, 8^9, \{y\})$ kritériumhoz tartozó szeletének kiszámításával szemléltetjük. A 6.4. ábrán a munkahalmaz tartalma és annak változása, valamint a dinamikus szelet előállításának folyamata látható. A 0-ik iterációs lépés az inicializálást jelenti.

Előrefelé számolt előremutató

A 6.5. ábrán látható, előremutató szeleteket számító igényvezérelt algoritmus a kritérium által meghatározott változó előremutató függőségeit változókat megjelölve (és a megjelöléseket törölve) számítja. Ha az algoritmus futása során egy adott ponton egy változó meg van jelölve, az azt jelenti, hogy az adott változó „él”, azaz a legutóbbi definíciójában felhasználtunk legalább egy élő változót. Kezdetben az egyetlen élő változó az a – kritérium által meghatározott – változó, amelynek az előremutató szeletét ki akarjuk számolni, ezért először ezt jelöljük meg (3. sor). Ezután egy ciklus dolgozza fel

iterációs lépés	munkahalmazból kivett k^l akció	az S szelet az iterációs lépés végén	munkahalmazhoz hozzáadott akciók	munkahalmaz az iterációs lépés végén
0	—	$\emptyset$	8^9	$\{8^9\}$
1	8^9	$\emptyset$	5^5	$\{5^5\}$
2	5^5	$\{5\}$	4^4, 3^3	$\{4^4, 3^3\}$
3	4^4	$\{4, 5\}$	1^1	$\{3^3, 1^1\}$
4	3^3	$\{3, 4, 5\}$	—	$\{1^1\}$
5	1^1	$\{1, 3, 4, 5\}$	—	$\emptyset$

6.4. ábra. Példa: Algorithm-DB egy lefutása

a végrehajtási nyomot a kritériumban szereplő i^j akciótól kezdve. Ez a ciklus vagy az *EH* végének elérésével, vagy a megjelölt változók elfogyásával ér véget. Ha az éppen feldolgozás alatt álló utasításban felhasználunk legalább egy élő változót (8. sor), akkor az itt definiált változót szintén megjelöljük (9. sor), és az utasítás a keresett S szelet részévé válik, hiszen – ha csak tranzitívan is – felhasználjuk a kritériumban definiált változó értékét. A 11. sorra azért van szükség, mert ha egy (potenciálisan) élő változót úgy definiálunk újra, hogy ahhoz semmilyen élő változót nem használunk fel, akkor a változó halottá válik.

Az algoritmus működését a 6.1. ábrán látható program $CF = (\langle a = 1\rangle, 3^3)$ kritériumhoz tartozó szeletének kiszámításával szemléltetjük. A 6.6. ábrán a megjelölt változókat és a jelölések változását, valamint a dinamikus szelet előállításának folyamatát láthatjuk. A 0-ik iterációs lépés az inicializálást jelenti.

6.2.2. Praktikus algoritmusok

Egy igényvezérelt algoritmusok – a program függőségeinek struktúrájától függően – a végrehajtási nyom egy jelentős részét többször is feldolgozhatják, mialatt több szelet kiszámítását egymástól függetlenül elvégzik. Igaz ugyan, hogy az algoritmusok csak a szelethez tartozó függőségeket fogják végigkövetni, de eközben a többi akción is át kell haladniuk. Ha ehhez hozzávesszük, hogy a szeletelés több alkalmazási területén is szükség lehet egyszerre több szeletre, akkor ésszerűnek tűnik, hogy az *EH* egyszeri feldolgozásával tudjunk több szeletet is kiszámolni. Ez nyilvánvalóan több számítást igényel, de egy bizonyos számú kritérium felett jobban megéri egy globális, minden dinamikus szeletet kiszámító algoritmust futtatni, mint egyenként kiszámolni a szeleteket. Kétféleképpen lehet ilyen algoritmusokat készíteni. Az egyik lehetőség, hogy a szelet és az *EH* feldolgozásának iránya megegyezik, azaz tulajdonképpen párhuzamosan számolunk az egyedi

```
        program Algorithm-DF(P, CF)
        input: P : a program
                CF = (x, i^j) : a dinamikus szeletelési kritérium
        output: S : a P program CF kritériumhoz tartozó szelete
        begin
 1:         S := ∅
 2:         Read(EH)
 3:         mark(d(i))
 4:         l := j
 5:         while ∃ megjelölt változó and l < J do
 6:             l := l + 1
 7:             k := EHI(l)
 8:             if ∃(u ∈ U(k) and marked(u)) then
 9:                 mark(d(k))
10:                 S := S ∪ {k}
               else
11:                 unmark(d(k))
               endif
            endwhile
12:         Output(S): a CF kritériumhoz tartozó szelet
        end
```

6.5. ábra. Algorithm-DF: igényvezérelt algoritmus előremutató szeletekhez

iterációs lépés	feldolgozott k^l akció	iterációs lépésben megjelölt változók	iterációs lépésben törölt megjelölések	megjelöltek halmaza az iterációs lépés végén	az S szelet az iterációs lépés végén
0	3^3	x	—	$\{x\}$	$\emptyset$
1	4^4	—	p	$\{x\}$	$\emptyset$
2	5^5	y	—	$\{x, y\}$	$\{5\}$
3	6^6	—	x	$\{y\}$	$\{5\}$
4	7^7	—	a	$\{y\}$	$\{5\}$
5	4^8	—	p	$\{y\}$	$\{5\}$
6	8^9	z	—	$\{y, z\}$	$\{5, 8\}$

6.6. ábra. Példa: Algorithm-DF egy lefutása

szeleteket. A párhuzamos számítás miatt itt is sokat nyerünk az egyedi szeletszámítások összességéhez képest, de a kritériumokhoz tartozó szeleteket tároló adatstruktúrát a szeletelés elejétől a végéig fent kellene tartanunk. Az ilyenfajta *párhuzamos* algoritmusokat a 6.2.3. fejezetben mutatjuk be.

Szerencsére lehetséges olyan praktikus algoritmusok megalkotása amelyeknél az algoritmus futása során nincs szükség az összes szelet tárolására, csak az egyes változókhoz tartozó függőségi halmazok karbantartására, és ezekből összeállítható az összes szelet. Ezek a függőségi halmazok utasítás sorszámokat tartalmaznak, mégpedig a végrehajtás adott pontján az adott változók aktuális dinamikus függőségeit. Ezeket a függőségi halmazokat a $d : U$ információk alapján kapjuk, és minden végrehajtási lépésben folyamatosan karbantartjuk. Ezzel a módszerrel képesek vagyunk pusztán ezen halmazok aktuális értékei alapján dinamikus szeleteket számolni minden lehetséges dinamikus szeletelési kritériumhoz. A megközelítés érdekes kettőssége, hogy az említett halmazok előállításához adott irányú szeletelés esetén a végrehajtási nyomot fordított irányban kell feldolgozni, azaz hátramutató szeletelés esetén előrefelé, előremutató szeletelés esetén hátrafelé. De bármennyire is furcsának tűnik, ez így természetes, hiszen míg a hátramutató függőségek egy adott végrehajtási ponton akkor válnak elérhetővé ha a nyom ez előtti részét már feldolgoztuk, ez pedig előrehaladó feldolgozás esetén történik meg, addig az előremutató függőségeket az adott végrehajtási pont utáni rész feldolgozásával kapjuk, ami visszafelé haladó EH feldolgozás esetén fog az adott pont feldolgozása előtt megtörténni.

A hátramutató szeleteket előrehaladó feldolgozással számító globális algoritmus a gyakorlatban leginkább használható algoritmus, mert egyszerre az összes (hátramutató) szeletet kiszámítja, és a végrehajtási történetet a természetes előrehaladó irányban dolgozza fel, ezáltal lehetőséget teremt arra, hogy az EH-t azonnal, a program végrehajtásával párhuzamosan feldolgozzuk. Az előremutató szeleteket számító algoritmust az a tény teszi kevésbé használhatóvá, hogy a végrehajtási nyom visszafelé irányú feldolgozása a gyakorlatban nem minden esetben triviális probléma.

Előrefelé számolt hátramutató

Ez az algoritmus Gyimóthy Tibor és társainak eredménye [28], de a teljesség kedvéért ezt is bemutatjuk. Az alapötlet az, hogy amikor a feldolgozás során elérünk egy akciót, ennek a hátramutató szeleteit ki tudjuk számolni az előzőleg kiszámolt szeletek és az akció direkt (közvetlen) függőségei alapján, amiket az adott pontban használt változók határoznak meg. A kiszámolt szeleteket azután tároljuk, de csak addig, amíg valóban szükség van rájuk.

```
      program Algorithm-GB(P, x)
      input: P : a program
             x : a program input
      output: hátramutató szelet az összes CB_j = (x, i^j, U(i))
                    kritériumhoz (j = 1 ... J)
      begin
1:        Inicializáljunk minden DynDep halmazt ∅ értékkel
2:        Read(EH)
3:        for j = 1 to J do
4:            i := EHI(j)
5:            DynDep(d(i)) := ⋃_{u_k ∈ U(i)} (DynDep(u_k) ∪ {LS(u_k)})
6:            LS(d(i)) := i
7:            Output(DynDep(d(i))): a CB_j kritériumhoz tartozó szelet
          endfor
      end
```

6.7. ábra. Algorithm-GB: globális algoritmus hátramutató szeletekhez

A hátramutató szeleteket előrefelé számoló globális algoritmus formalizálva a 6.7 ábrán látható. A végrehajtási nyom feldolgozása az első akció feldolgozásával kezdődik. Az algoritmus a j-ik lépésben a $CB_j = (\mathbf{x}, i^j, U(i))$ kritériumhoz tartozó szeletet számítja ki, és tárolja el a $DynDep(d(i))$ halmazban. Az 5. sorban számolt CB_j-hez tartozó hátramutató szelet a j-ik akcióban használt változókat utoljára definiáló utasítások halmazának, és a változókat utoljára definiáló akciókhoz tartozó hátramutató szeleteknek az uniója. Ez utóbbiakat az adott pillanatban a megfelelő *DynDep* halmazok tárolják. Mivel a további szeletek számításához csupán a változókat utoljára definiáló akciók szeleteire van szükség, a *DynDep* halmazok nem az akciókhoz, hanem a változókhoz köthetőek. Az új szelet meghatározásával az új értékek kiszámításának sorrendje miatt $d(i) \in U(i)$ (önmagától függő változók) esetében sincsenek problémák.

Ezzel az algoritmussal a végrehajtási nyomban szereplő összes i^j akcióhoz tartozó $CB_j = (\mathbf{x}, i^j, U(i))$ kritérium hátramutató dinamikus szeleteit megkapjuk az *EH* egyszeri végigolvasásával, mégpedig már az adott akció feldolgozásának végén, így a szeletek eltárolására nincs szükség, azok egyből kiírhatók.

Az algoritmus működése a 6.1. ábrán látható programon $\mathbf{x} = \langle a = 1 \rangle$ input mellett a 6.8. ábrán látható. Az ábrán az éppen feldolgozott akció mellett az akcióban definiált változó ($d(i)$), a CB_j kritériumhoz kiszámított szelet – ami egyben a változóhoz tartozó $DynDep(d(i))$ halmaz új értéke is – és a változót utoljára definiáló utasítás ($LS(d(i))$) új sorszáma látható.

feldolgozott i^j akció	$d(i)$	CB_j szelete: $DynDep(d(i))$ a lépés végén	$LS(d(i))$
1^1	a	$\emptyset$	1
2^2	y	$\emptyset$	2
3^3	x	$\emptyset$	3
4^4	p	$\{1\}$	4
5^5	y	$\{1,3,4\}$	5
6^6	x	$\{1,4\}$	6
7^7	a	$\{1,4\}$	7
4^8	p	$\{1,4,7\}$	4
8^9	z	$\{1,3,4,5\}$	8

6.8. ábra. Példa: Algorithm-GB egy lefutása

Visszafelé számolt előremutató

A előremutató szeleteket előállító visszafelé számoló globális algoritmus működése nagyban hasonlít a hátramutató szeleteket globálisan számító algoritmus működésére, csak bizonyos szempontból annak éppen a fordítottja. Az algoritmus elve most is az, hogy az éppen feldolgozott akció szeleteit az akciótól közvetlenül függő végrehajtási pontok szeleteiből számítjuk. A nyom fordított feldolgozása miatt az akció feldolgozásakor itt is a végleges szeletét kapjuk meg. Csakúgy mint a másik globális algoritmus esetében, itt is csak a szükséges ideig tároljuk a kiszámolt szeleteket.

Az algoritmust formalizálva a 6.9. ábrán láthatjuk. A végrehajtási nyom feldolgozása az utolsó akcióval kezdődik. Az algoritmus minden v változóhoz hozzárendel egy $LiveAt(v)$ halmazt, ami azoknak a már feldolgozott akcióknak az utasítás sorszámait tartalmazza, amelyeknek a definiált változói függenek a v változót utoljára definiáló (még fel nem dolgozott) akciótól. Így minden lépés elején az éppen feldolgozott i^j akcióban definiált $d(i)$ változó $LiveAt(d(i))$ halmaza pontosan a $CF_j = (\mathbf{x}, i^j)$ kritériumhoz tartozó előremutató szeletet fogja tartalmazni. Az algoritmus 7. sorában az éppen feldolgozott akció által használt $U(i)$ változók $LiveAt$ halmazait kibővítjük az aktuális akció utasításával (i) és a definiált $d(i)$ változótól függő utasítások halmazával (amit a $LiveAt(d(i))$ halmazban tárolunk), mert minden $d(i)$-től függő utasítás függ az $U(i)$ halmaz összes elemétől is. Ha $d(i)$ nem függ önmagától, akkor a $LiveAt(d(i))$ halmazt ki kell ürítenünk (9. sor), hiszen – az éppen feldolgozott akció alapján – a változó ezt megelőző definícióját még sehol sem használtuk fel, így attól semmi sem függ.

Csakúgy mint az előző algoritmusnál, itt is az összes i^j akcióhoz már az akció fel-

```
program Algorithm-GF(P, x)
input: P : a program
       x : a program input
output: előremutató szelet az összes CF_j = (x, i^j)
        kritériumhoz (j = 1 ... J)
begin
    Inicializáljunk minden LiveAt halmazt ∅ értékkel
    ReadStore(EH)
    for j = J downto 1 do
        i := EHI(j)
        Output(LiveAt(d(i))): a CF_j kritériumhoz tartozó szelet
        for u_k ∈ U(i) do
            LiveAt(u_k) := LiveAt(u_k) ∪ LiveAt(d(i)) ∪ {i}
        endfor
        if d(i) ∉ U(i) then
            LiveAt(d(i)) := ∅
        endif
    endfor
end
```

6.9. ábra. Algorithm-GF: globális algoritmus előremutató szeletekhez

dolgozásakor megkapjuk a $CF_j = (\mathbf{x}, i^j)$ kritériumhoz tartozó előremutató dinamikus szeletet, tehát a szeletek külön tárolására itt sincs szükség.

Az algoritmus működését a 6.1. ábrán látható programon $\mathbf{x} = \langle a = 1 \rangle$ inputra a 6.10. ábrán szemléltetjük. Az ábrán a feldolgozott akció mellett a megfelelő utasítás $d : U$ reprezentációja, a kiszámolt szelet illetve azok a változók és *LiveAt* halmazaik új értéke látható, amely változókhoz tartozó *LiveAt* halmazok értéke (potenciálisan) megváltozott a lépés során.

6.2.3. Párhuzamos algoritmusok

A két igényvezérelt algoritmus a szeletelés irányával megegyező irányban dolgozza fel a végrehajtási nyomot. Ugyanakkor a 6.2.2. alfejezetben megismert két globális algoritmusnál a szeletelés és az *EH* feldolgozásának iránya fordított. Az utóbbiaknak megvan az az előnye, hogy az adott végrehajtási lépéshez tartozó végső szeletet már az adott akció feldolgozásakor kiszámolja. A felosztásunk szerint két további globális algoritmus lehetséges, amelyekben – csakúgy, mint az igényvezérelt algoritmusokban – a szeletek és a nyom feldolgozásának iránya megegyezik. Ezeket mi *párhuzamos globális* algoritmusoknak nevezzük, mert a szeleteket párhuzamosan számolják – mintha sok igényvezérelt algoritmust futtatnánk párhuzamosan –, és a végrehajtási nyom teljes feldolgozásáig potenciálisan egyetlen szelet sem tekinthető véglegesnek. De ezeknek az algoritmusoknak is

feldolgozott i^j akció	$d(i)$: $U(i)$	CF_j szelete: $LiveAt(d(i))$ a lépés kezdetén	változott halmazok v: $LiveAt(v)$
8^9	z: $\{y\}$	$\emptyset$	y: $\{8\}$
			z: $\emptyset$
4^8	p: $\{a\}$	$\emptyset$	a: $\{4\}$
			p: $\emptyset$
7^7	a: $\{a, p\}$	$\{4\}$	a: $\{4, 7\}$
			p: $\{4, 7\}$
6^6	x: $\{p\}$	$\emptyset$	p: $\{4, 6, 7\}$
			x: $\emptyset$
5^5	y: $\{x, p\}$	$\{8\}$	x: $\{5, 8\}$
			p: $\{4, 5, 6, 7, 8\}$
			y: $\emptyset$
4^4	p: $\{a\}$	$\{4, 5, 6, 7, 8\}$	a: $\{4, 5, 6, 7, 8\}$
			p: $\emptyset$
3^3	x: $\emptyset$	$\{5, 8\}$	x: $\emptyset$
2^2	y: $\emptyset$	$\emptyset$	y: $\emptyset$
1^1	a: $\emptyset$	$\{4, 5, 6, 7, 8\}$	a: $\emptyset$

6.10. ábra. Példa: Algorithm-GF egy lefutása

megvan az előnye azzal szemben, hogy valóban az igényvezérelt algoritmusokkal számítjuk ki az összes szeletet. Nevezetesen, az egyes akcióknál fellépő közvetlen függőségeket csupán egyszer kell kiszámolnunk (ezen az információn megosztozik a sok párhuzamos szelet-számítás), így a végrehajtási nyomot is elegendő egyszer bejárnunk.

Bár az előrehaladó párhuzamos algoritmusnak nagyobb a memóriaigénye mint az előremutató szeleteket visszafelé számító globális algoritmusnak, a végrehajtási nyomot a természetes előrehaladó irányban feldolgozó működése miatt előbbi praktikus alternatívája lesz az utóbbinak. Ugyanez nem mondható el a hátrahaladó párhuzamos algoritmusról. A nagyobb memóriaigény és az *EH* visszafelé irányú feldolgozása miatt a hátrahaladó párhuzamos algoritmus nem igazán praktikus alternatíva, de a teljesség kedvéért ezt is megadjuk.

Mint látni fogjuk, a két párhuzamos algoritmus a globális algoritmusokhoz nagyon hasonló módon működik. Olyannyira, hogy majdnem ugyanazokat a *DynDep* és *LiveAt* halmazokat használják, de lényeges különbség, hogy ezekben utasítások helyett akciókat tárolnak. Utasítások helyett akciókat tárolni viszont meglehetősen költséges. De ha megelégszünk kevésbé precíz szeletekkel is, akkor az eredeti (utasításokat tároló) *DynDep* és *LiveAt* halmazok használatával csökkenthetjük a költségeinket azon az áron, hogy a hasonló akciókhoz egy összegzett függőség-halmazt tárolunk, tehát tulajdonképpen statikus kritériumokra számítunk kvázi-dinamikus szeleteket.

Visszafelé számolt hátramutató

A 6.11. ábrán látható, a hátramutató szeleteket a végrehajtási nyom visszafelé irányú feldolgozásával számoló algoritmus működési elve, hogy az i^j akció feldolgozásakor az i^j-től függő akciók szeletei kibővíthetőek i-vel. Az algoritmus végrehajtása során folyamatosan karbantartunk egy *LiveAt* adatstruktúrát, ami a szükséges előremutató függőségeket tartalmazza minden egyes definiált változóhoz.

Mivel a függőségek követésének ugyanaz a logikája mint az előremutató szeleteket visszafelé számító globális algoritmusnak, az két algoritmus lényegi része nagyon hasonlít egymáshoz. A fő különbség azonban, hogy itt akciókat tárolunk a *LiveAt* halmazokban utasítások helyett. A szeleteket az iterációs lépések elején bővítjük: az iterációs lépésben definiált $d(i)$ változótól függő akciók szeleteit kibővítjük az aktuális programsorral (i). Az 5. sorban a $LiveAt(d(i))$ pontosan ezeket az akciókat tartalmazza. A szeleteket akkor írjuk ki az, ha a teljes végrehajtási nyom feldolgozását befejeztük.

Nyilvánvalóan előfordulhat, hogy egyetlen szeletet sem tudunk lezárni a nyom utoljára feldolgozott akciójának feldolgozása előtt. Bár több számítás vagy adatstruktúra árán egy adott szeletről megállapítható lenne, hogy bővülhet-e még további akciók fel-

```
      program Algorithm-PB(P, x)
      input: P : a program
             x : a program input
      output: hátramutató szelet az összes CB_j = (x, i^j, U(i))
              kritériumhoz (j = 1 ... J)
      begin
 1:       Inicializáljunk minden S és LiveAt halmazt ∅ értékkel
 2:       ReadStore(EH)
 3:       for j = J downto 1 do
 4:           i := EHI(j)
 5:           for a_k ∈ LiveAt(d(i)) do
 6:               S(a_k) := S(a_k) ∪ {i}
              endfor
 7:           for u_k ∈ U(i) do
 8:               LiveAt(u_k) := LiveAt(u_k) ∪ LiveAt(d(i)) ∪ {i^j}
              endfor
 9:           if d(i) ∉ U(i)then
10:               LiveAt(d(i)) := ∅
              endif
          endfor
11:       for ∀ i^j ∈ EH do
12:           Output(S(i^j)): a CB_j kritériumhoz tartozó szelet
          endfor
      end
```

6.11. ábra. Algorithm-PB: párhuzamos algoritmus hátramutató szeletekhez

dolgozása során.

Az algoritmus lefutását $\mathbf{x} = \langle a = 1 \rangle$ inputra a 6.1. ábrán látható példaprogramon a 6.12. ábrán szemléltetjük. Az ábrán az akció és a hozzá tartozó $d : U$ reprezentáció mellett látható a $LiveAt(d(i))$ halmaz kezdeti értéke az adott lépésben, felsoroltuk a lépés során potenciálisan megváltozott szeleteket (és, hogy melyik akcióhoz tartoznak) és *LiveAt* halmazokat is.

Előrefelé számolt előremutató

A 6.13. ábrán látható, előremutató szeleteket a végrehajtási nyom előrehaladó feldolgozásával számító algoritmus működési elve a következő. Egy adott iterációs lépésben, melyben az i^j akciót dolgozzuk fel, az i utasítással kibővíthetjük azon akciók szeleteit, amelyektől az i^j akció függ. Az algoritmus végrehajtása során folyamatosan karbantartunk egy *DynDep* adatstruktúrát, ami a szükséges hátramutató függőségeket tartalmazza minden egyes definiált változóhoz. Hasonlóan a többi globális algoritmushoz, ezeket a függőségeket is elegendő a változókhoz rendelve tárolni.

Az algoritmus első része nagyon hasonlít a hátramutató szeleteket előrefelé számító globális algoritmushoz. A fő különbség itt is – csakúgy mint a visszafelé számolt globális algoritmusok között – az, hogy a *DynDep* halmazokban nem utasításokat hanem akciókat tárolunk. Az iterációs lépés végén azután (8. sor) az aktuális utasítás i sorszámával bővítjük azoknak az akcióknak a szeleteit amelyektől a definiált változó függ. A szeleteket csak a teljes *EH* feldolgozása után írjuk ki.

Csakúgy mint a hátramutató párhuzamos algoritmus esetében, potenciálisan egyetlen egy szeletet sem tekinthető véglegesnek a végrehajtási nyom teljes feldolgozása előtt, bár plusz számításokkal és/vagy plusz adatok tárolásával ebben az esetben is meghatározható lenne, hogy melyek azok a szeletek, amelyek az adott lépés után már biztosan nem bővülnek.

Az algoritmus működését az $\mathbf{x} = \langle a = 1 \rangle$ inputtal és a 6.1. ábrán látható példaprogrammal a 6.14. ábrán szemléltetjük. A definiált változó mellett feltüntettük a *DynDep* halmazának és az őt definiáló utolsó akció új (az adott iterációban) kiszámított értékét, valamint az iterációban megváltozott szeleteket is.

6.3. DDG ekvivalencia

Ahhoz, hogy megmutassuk, hogy az általunk használt $d : U$ alapú algoritmus ugyanazokat a szeleteket számolja mint a DDG alapú algoritmus, először a két reprezentáció elemeinek ekvivalenciáját kell megmutatnunk. Ez az ekvivalencia a definíciókból ered.

i^j	$d(i)$: $U(i)$	$LiveAt(d(i))$ a lépés kezdetén	változott szeletek a: $S(a)$	változott függőségek v: $LiveAt(v)$
8^9	z: $\{y\}$	$\emptyset$	—	y: $\{8^9\}$
				z: $\emptyset$
4^8	p: $\{a\}$	$\emptyset$	—	a: $\{4^8\}$
				p: $\emptyset$
7^7	a: $\{a, p\}$	$\{4^8\}$	4^8: $\{7\}$	a: $\{7^7, 4^8\}$
				p: $\{7^7, 4^8\}$
6^6	x: $\{p\}$	$\emptyset$	—	p: $\{6^6, 7^7, 4^8\}$
				x: $\emptyset$
5^5	y: $\{x, p\}$	$\{8^9\}$	8^9: $\{5\}$	x: $\{5^5, 8^9\}$
				p: $\{5^5, 6^6, 7^7, 4^8, 8^9\}$
				y: $\emptyset$
4^4	p: $\{a\}$	$\{5^5, 6^6, 7^7, 4^8, 8^9\}$	5^5: $\{4\}$	a: $\{4^4, 5^5, 6^6, 7^7, 4^8, 8^9\}$
			6^6: $\{4\}$	p: $\emptyset$
			7^7: $\{4\}$	
			4^8: $\{4, 7\}$	
			8^9: $\{4, 5\}$	
3^3	x: $\emptyset$	$\{5^5, 8^9\}$	5^5: $\{3, 4\}$	x: $\emptyset$
			8^9: $\{3, 4, 5\}$	
2^2	y: $\emptyset$	$\emptyset$	—	y: $\emptyset$
1^1	a: $\emptyset$	$\{4^4, 5^5, 6^6, 7^7, 4^8, 8^9\}$	4^4: $\{1\}$	a: $\emptyset$
			5^5: $\{1, 3, 4\}$	
			6^6: $\{1, 4\}$	
			7^7: $\{1, 4\}$	
			4^8: $\{1, 4, 7\}$	
			8^9: $\{1, 3, 4, 5\}$	

6.12. ábra. Példa: Algorithm-PB egy lefutása

```
program Algorithm-PF(P, x)
input: P : a program
       x : a program input
output: előremutató szelet az összes CF_j = (x, i^j)
        kritériumhoz (j = 1 ... J)
begin
    Inicializáljunk minden S és DynDep halmazt ∅ értékkel
    Read(EH)
    for j = 1 to J do
        i := EHI(j)
        DynDep(d(i)) := ⋃_{u_k ∈ U(i)} (DynDep(u_k) ∪ {LA(u_k)})
        LA(d(i)) := i^j
        for a_k ∈ DynDep(d(i)) do
            S(a_k) := S(a_k) ∪ {i}
        endfor
    endfor
    for ∀ i^j ∈ EH do
        Output(S(i^j)): a CF_j kritériumhoz tartozó szelet
    endfor
end
```

6.13. ábra. Algorithm-PF: párhuzamos algoritmus előremutató szeletekhez

Kezdjük a statikus reprezentációval. Legyenek a program utasításai az $i \in \{1, \ldots, I\}$ értékekkel azonosítva. A PDG (*Procedure Dependency Graph*, a már említett SDG egy alkotóeleme, egy eljárást reprezentál) a program minden egyes i utasításához hozzárendel egy P_i gráfpontot, a $d : U$ reprezentáció pedig egy $d_i : U_i$ adatpárt. Mivel mindkét hozzárendelés bijektív, P_i és $d_i : U_i$ egyértelműen megfeleltethető egymásnak. A PDG-ben létezik $P_i \rightarrow P_k$ vezérlési függőségi él, ha k vezérlési függésben van i-től[2]. Ez akkor lehetséges, ha i egy döntési utasítás, akkor viszont a $d : U$ reprezentációban d_i egy predikátumváltozó, és definíció szerint $d_i \in U_k$. Ha d_i egy predikátumváltozó és $d_i \in U_k$, akkor a definíció alapján létezik a $P_i \rightarrow P_k$ vezérlési függőségi él. Ha a PDG tartalmazza a $P_i \rightarrow P_k$ adatfüggőségi élet, az azt jelenti, hogy a P_i-ben kiszámolt értéket P_k-ban felhasználjuk, vagyis az i-ik utasításban definiált változót a k-ik utasításban felhasználjuk, vagyis $d_i \in U_k$. Adatfüggőségnél a fordított irányú megfelelés nem egyértelmű, mert a statikus $d : U$ reprezentáció a vezérlési folyam információktól nem függ, tehát nem biztos, hogy ha d_i eleme is U_k-nek, létezik a $P_i \rightarrow P_k$ adatfüggőségi él, viszont a dinamikus reprezentációk között ez az irány is belátható.

Most tehát nézzük a DDG gráf és az *EH*-val kombinált $d : U$ reprezentáció ek-

[2]Bár Agrawal eredeti terminológiája szerint az él a függő pont felől mutat afelé, amitől az függ, a későbbi irodalmak nagy része már az általunk most is használt, az eredeti terminológiával ellentétes irányú élekkel dolgozik. Valójában az élek iránya konzisztens használat mellett teljesen lényegtelen.

i^j	$d(i)$	$DynDep(d(i))$ új értéke	$LA(d(i))$ új értéke	változott szeletek a: $S(a)$
1^1	a	$\emptyset$	1^1	—
2^2	y	$\emptyset$	2^2	—
3^3	x	$\emptyset$	3^3	—
4^4	p	$\{1^1\}$	4^4	1^1: $\{4\}$
5^5	y	$\{1^1, 3^3, 4^4\}$	5^5	1^1: $\{4, 5\}$
				3^3: $\{5\}$
				4^4: $\{5\}$
6^6	x	$\{1^1, 4^4\}$	6^6	1^1: $\{4, 5, 6\}$
				4^4: $\{5, 6\}$
7^7	a	$\{1^1, 4^4\}$	7^7	1^1: $\{4, 5, 6, 7\}$
				4^4: $\{5, 6, 7\}$
4^8	p	$\{1^1, 4^4, 7^7\}$	4^8	1^1: $\{4, 5, 6, 7\}$
				4^4: $\{4, 5, 6, 7\}$
				7^7: $\{4\}$
8^9	z	$\{1^1, 3^3, 4^4, 5^5\}$	8^9	1^1: $\{4, 5, 6, 7, 8\}$
				3^3: $\{5, 8\}$
				4^4: $\{4, 5, 6, 7, 8\}$
				5^5: $\{8\}$

6.14. ábra. Példa: Algorithm-PF egy lefutása

vivalenciáját. Legyen adott egy $EH = \langle i_1^1, \ldots, i_J^J \rangle$ végrehajtási nyom (ez független a program reprezentációtól). A DDG-ben minden $i^j \in EH$ végrehajtási lépésnek egyértelműen megfelel egy P_{i^j} pont. Definíció szerint akkor létezik $P_{i^j} \rightarrow P_{k^l}$ vezérlési függőségi él, ha létezik $P_i \rightarrow P_k$ vezérlési függőségi él (PDG-ben) és az l-ik lépésben végrehajtott k utasítás a j-ik lépésben végrehajtott i utasítástól függ, azaz a kettő között nem lett végrehajtva másik i utasítás. A $P_i \rightarrow P_k$ él miatt $d_i \in U_k$, és mivel i^j és k^l között nem lett végrehajtva i, és d_i predikátumváltozót csak i definiálja, a $LD(d_i, l) = i^j$. Hasonlóképpen, ha $k^l \in EH$ és a $p \in U_k$ predikátumváltozóra $LD(p, l) = i^j$, akkor $p = d_i$ és k (statikusan) kontrol-függ i-től, és mivel i utoljára a j-ik lépésben lett végrehajtva (ami p egyediségéből és $LD(p, l) = i^j$-ből adódik), ezért k^l az i^j-től függ, tehát létezik a $P_{i^j} \rightarrow P_{k^l}$ él. Ha létezik a $P_{i^j} \rightarrow P_{k^l}$ adatfüggőségi él, az azt jelenti, hogy a k utasítás felhasználja az i-ben kiszámolt értéket, azaz $d_i \in U_k$, továbbá i^j és k^l között nem lett végrehajtva olyan i' utasítás, ami szintén az i-ben definiált változót definiálja, vagyis $LD(d_i, l) = i^j$. Ha $k^l \in EH$, $v \in U_k$ és $LD(v, l) = i^j$, az azt jelenti, hogy $v = d_i$, és mivel i^j és k^l között nem volt végrehajtva olyan i' utasítás, ami a $v = d_i$ változót definiálta volna, a k^l adat-függ i^j-től, azaz létezik a $P_{i^j} \rightarrow P_{k^l}$ él.

A fent leírtak alapján tehát akkor és csak akkor létezik $i^j \rightarrow k^l$ él a gráfban, ha $d_i \in U_k$ és $LD(d_i, l) = i^j$.

6.3.1. Igényvezérelt algoritmusok

Igényvezérelt algoritmusok esetén az eredmény ekvivalenciáját úgy mutatjuk meg, hogy a leírt algoritmust a DDG gráfpontok valamint a $d : U$ változók és akciók ekvivalenciájából adódó átalakításokkal egy gráfszínezési algoritmussá alakítjuk, amely egy adott kiindulópontból az élek mentén elérhető[3] gráfpontokat fogja beszínezni, amikhez tartozó utasítások definíció szerint pont a DDG-ből előállított szeletet alkotják. Ki fogjuk használni a gráf azon tulajdonságát, hogy egy irányított körmentes gráf, és i^j címkéjű pontjait j szerint rendezve nem lesz olyan $i^j \rightarrow k^l$ cél, amire $l < j$.

Visszafelé haladó hátramutató algoritmus

Az algoritmus S-ben gyűjti a kiszínezett gráfpontokhoz tartozó utasításokat, ez a halmaz kezdetben üres (1. sor). A *worklist* tárolja a színezendő de még fel nem dolgozott pontokat, ide először az i^j kezdőpont kerül (3. sor). A ciklus mindaddig fut, amíg van színezendő de még színezetlen pont (4. sor). A ciklusban kiszínezünk egy színezhető

[3] Az elérés minden esetben a szeletelésnek megfelelő gráfpont-elérést jelenti, azaz a szeletelés irányának függvényében a gráf éleit csak az irányuknak megfelelően (előremutató), vagy csak azzal ellentétesen (hátramutató) vesszük figyelembe.

k^l pontot (5. sor), majd S-be beletesszük k-t (7. sor), hiszen kiszínezett ponthoz tartozó utasítás. Majd vesszük az összes k-ba érkező élet, és a kezdőpontjaikat színezendőként megjelöljük (8-9. sor). Az algoritmus ki fogja színezni az i^j kezdőpontból elérhető pontokat, és az ezekhez tartozó utasításokat – tehát a kritériumhoz tartozó szeletet – gyűjti össze S-be. Mint látható, a két algoritmus (a $d : U$ reprezentáción dolgozó szeletelő, és a DDG gráfon dolgozó színező) teljes mértékben ekvivalens, tehát ugyanazt a szeletet produkálják.

Előrefelé haladó előremutató algoritmus

Címkézzük fel a gráf minden k^l pontját d_k-val (is). Az algoritmus S-ben gyűjti a kiszínezett gráfpontokhoz tartozó utasításokat, ez a halmaz kezdetben üres (1. sor). A kezdőpont i^j, így jelöljünk meg minden d_i-vel címkézett pontot (3. sor). Induljunk ki az i^j pontból (4. sor), majd a ciklusban vegyük a soron következő pontot amit jelöljön k^l (6-7. sor). Ha megjelölt pontból fut él k^l-be (8. sor), akkor színezzük k^l-t és jelöljük meg a d_k-val címkézett pontokat (9. sor), különben töröljük a d_k-val címkézett pontok jelölését. Itt meg kell jegyeznünk, hogy ha létezik egy $k'^{l'} \rightarrow k^l$ él, akkor $LD(d_{k'}, l) = k'^{l'}$, és mivel ez volt a legutóbb feldolgozott $d_{k'}$-vel címkézett pont, akkor és csak akkor lesz k^l pont feldolgozásakor megjelölve, ha kiszíneztük. Ha az aktuális k^l pontot színeztük, akkor k bekerül S-be (10. sor), hiszen kiszínezett ponthoz tartozó utasítás. Az algoritmus minden i^j-ből elérhető pontot meg fog találni a gráf irányítottsága és a gráfpontok rendezettsége miatt, tehát S-ben a megfelelő szelet elemei gyűlnek össze. És itt is igaz, hogy a kétféle reprezentáción dolgozó algoritmus teljesen ekvivalens, tehát ugyanazt a szeletet számolja.

6.3.2. Praktikus algoritmusok

Annak bizonyítására, hogy a praktikus algoritmusok is a DDG alapú szeleteléssel kapott szeleteket számolják ki teljes indukciót alkalmazunk. Kezdetben a szeleteket tartalmazó *DynDep* és *LiveAt* halmazok üresek, ami az első akció feldolgozása előtt triviálisan helyes. Most tegyük fel, hogy az i^j akciót feldolgozó iteráció kezdetén az összes *DynDep* és *LiveAt* halmaz a megfelelő (DDG algoritmus által is kiszámolt) szeleteket vagy szelet-kezdeményeket tartalmazza. Megmutatjuk, hogy ezekből az értékekből az algoritmusok az i^j akció feldolgozása után a megfelelő szeleteket állítják elő. Ehhez fel fogjuk használni azt a nyilvánvaló tényt, hogy a gráfelérés alapú szelet alternatív módon is számolható. Ez annyit tesz, hogy ha a gráfban egy P_a pontból elérhető összes pontot összegyűjtjük, és vesszük az ezekhez tartozó utasításokat, ugyanazt kapjuk, mintha ezt

először (alternatív módon) megcsináljuk a P_a-ból közvetlenül elérhető pontokra, majd az így kapott utasításhalmazokat uniózzuk és hozzávesszük a közvetlenül elérhető pontokhoz tartozó utasításokat.

Előrefelé haladó hátramutató algoritmus

A fentiek hátramutató szeletelés esetén azt jelentik, hogy a *DynDep* halmazok a változókat utoljára definiáló akciók hátramutató szeleteit tartalmazzák, amik megegyeznek a DDG gráfban az adott (i^j előtti utolsó definícióknak megfelelő) pontokból elérhető pontokhoz tartozó utasításokkal. De az algoritmusunk – a gráfélek, használati halmazok és utolsó definíciók közötti összefüggések miatt – pontosan a fentebb említett alternatív szeletszámítás megvalósítása (5. sor), és a végeredmény szeletet $DynDep(d_i)$-ben tárolja, így – tekintve, hogy a lépés után d_i utolsó definíciója i^j lesz (6. sor) – az indukciós feltevés továbbra is fennáll.

Visszafelé haladó előremutató algoritmus

Előremutató szeletelés esetén a feltételezés az, hogy az i^j akció feldolgozásakor a *LiveAt* halmazok a megfelelő változót utoljára definiáló a akciók előremutató szeleteinek azon részét tartalmazzák, ami az *EH* szerint P_{i^j} mögötti és P_a-ból közvetlenül elérhető pontokon keresztül érhető el. Nyilvánvaló, hogy így a $LiveAt(d_i)$ halmaz az i^j feldolgozásakor az i^j teljes előremutató szeletét tartalmazza, hiszen nincs olyan él a gráfban, ami (tetszőleges) P-ből P előtti pontra mutatna. A megfelelő szeletek bővítése itt is az előzőleg említett alternatív szeletszámítással ekvivalens módon történik. Megnézzük, hogy mely pontokból van közvetlen él P_{i^j}-be (a felhasznált változókat utoljára definiáló akciók pontjai), és ezen pontok szeleteit (amiket az indukciós feltevés szerint éppen az adott változók *LiveAt* halmazai tartalmaznak) bővítjük az i^j szeletével és utasításával (6-7. sor). Az egyetlen speciális technikai művelet, hogy ha a fentiek alapján az „új" $LiveAt(d_i)$ halmaznak tartalmaznia kell a „régi" $LiveAt(d_i)$ halmazt, akkor egyszerűen nem töröljük, egyébként viszont – mivel $P_{LD(d_i,j)}$-ből nem lesz P_{i_j} mögé nyúló él – kiürítjük. Ezzel i^j feldolgozása után továbbra is fennáll az indukciós feltevés.

6.3.3. Párhuzamos algoritmusok

A párhuzamos algoritmusok DDG ekvivalenciájának bizonyítása nagyon hasonlít az azonos irányban haladó praktikus algoritmusnál alkalmazott bizonyításhoz. Az eltérés az, hogy a megfelelő *DynDep* és *LiveAt* halmazok nem a szeleteket (azok kezdeményeit) tartalmazzák, hanem azokat a pontokat (azok egy részét), amelyekből elérhető az ép-

pen feldolgozott pont. A bizonyítás indukciós feltétele ennek megfelelően változik, de az indukciós lépés bizonyítása ezt a változást tekintetbe véve ugyanaz. A szelet kiszámításának helyessége pedig a feltétel teljesülése esetén nyilvánvaló, hiszen az aktuális ponthoz tartozó i utasítást azoknak és csak azoknak a pontoknak a szeleteibe tesszük bele, amelyekből a pont elérhető.

Visszafelé haladó hátramutató algoritmus

Az indukciós feltevés tehát az, hogy a $LiveAt(v)$ halmaz az i^j feldolgozási lépésében azokat a pontokat tartalmazza, amelyekből $P_{LD(v,j)}$ elérhető egy olyan P_a ponton keresztül, amelyből $P_{LD(v,j)}$ közvetlenül elérhető, és ami az EH szerint P_{ij} mögött található.

Előrefelé haladó előremutató algoritmus

Ebben az esetben pedig az indukciós feltevés, hogy a $DynDep(v)$ halmaz az i^j feldolgozási lépésében azokat a pontokat tartalmazza, amelyekből $P_{LD(v,j)}$ elérhető.

6.4. Összefoglalás

Ebben a fejezetben megadtunk három szempontot, ami alapján a $d : U$ alapú dinamikus szeletelési algoritmusokat kategorizálni lehet, majd a szempontok 2-2 lehetséges értékének kombinálásával vázoltunk összesen 8 lehetséges algoritmust. Ezek közül kettő gyakorlatilag értelmetlennek bizonyult. A maradék hathoz (ami közül kettőt már korábban is publikáltak szerzőtársaim) megadtunk egy-egy algoritmust, és megmutattuk róluk, hogy ugyanazokat a szeleteket számítják mint az etalonként elfogadott Dynamic Dependence Graph (DDG) alapú dinamikus szeletelési algoritmus.

7. fejezet

Szeletelő algoritmusok implementációja

Az előző fejezetben bemutatott algoritmusok – mint azt említettük is – csak elméleti algoritmusok, egyáltalán nem foglalkoznak azokkal a problémákkal, amelyek egy valós programozási nyelv esetén felmerülnek. Pedig ezeknek a problémáknak a megoldására feltétlenül szükség van ahhoz, hogy a szeletelés ne csupán egy elméleti módszer legyen, hanem egy valóban használható segédeszköz a különböző alkalmazási területeken. A szerzőtársaimmal sikerült megoldást adnunk bizonyos ilyen jellegű kérdésekre, így a 7.1. táblázatban látható nyelvekhez (C, Java) illetve környezetben (GDB) implementáltunk néhány algoritmust.

Az igényvezérelt hátramutató és praktikus hátramutató szeletelést C nyelvre implementáltuk [10], [BGS01]. A végrehajtási nyomot forráskód instrumentálás segítségével nyerüjk ki, a $d : U$ reprezentációt a forráskód statikus elemzésével. A szeletelő algoritmusokat önálló programokban implementáltuk, amelyek inputjai az EH és a $d : U$ voltak.

Algoritmus	C	Java	GDB
Igényvezérelt előremutató			
Igényvezérelt hátramutató	•		
Praktikus hátramutató	•	•	•
Praktikus előremutató			
Párhuzamos előremutató		$*$	
Párhuzamos hátramutató			

($*$ egyszerűsített verzió)

7.1. táblázat. A szeletelő algoritmusok megvalósított implementációi

A praktikus hátramutató algoritmust és a párhuzamos előremutató algoritmus egy egyszerűsített verzióját szerzőtársaim implementálták Java programokra [55]. Itt forráskód instrumentálás helyett a Java virtuális gépre hagytuk az *EH* elkészítését, majd az így keletkezett bájtkód szintű információkat vezettük vissza a forráskód-elemekre. A párhuzamos algoritmus nagy erőforrásigénye miatt egy egyszerűsített algoritmust implementáltunk, ami nem az egyes akciókhoz számolt szeleteket, hanem az egyes utasításokhoz oly módon, hogy folyamatosan összegezte a megfelelő akciók szeleteit.

Szintén szerzőtársaim eredménye a praktikus hátramutató algoritmus GDB [26] debuggerbe integrált verziója [11]. A $d : U$ információkról egy módosított GCC fordító gondoskodik, míg a végrehajtási történetet maga a GDB produkálja. Az algoritmus szintén a GDB-ben van megvalósítva. Ez az implementáció nyelvfüggetlen.

A továbbiakban először megadunk néhány lehetséges problémát, amivel annak kell megküzdenie aki a fenti algoritmusokat egy valódi programozási nyelvre szeretné implementálni. Majd bemutatjuk, hogy az algoritmusok C nyelvre történő adaptálása és implementációja során a felmerült problémákat milyen módon oldottuk meg. Ezek közül a megoldások közül a változók és mutatók kezelésében a saját munkám volt meghatározó. Bemutatjuk továbbá a C nyelvre specializált praktikus hátramutató algoritmust, amelyben az említett változókezelésen kívül szerzőtársaim munkája volt meghatározó. Mivel a 8.2.2. fejezetben a Java implementációval végzett méréseinket is közöljük, ezt az implementációt is bemutatjuk, ami szerzőtársaim eredménye.

7.1. Valódi programozási nyelvek szeletelésének problémái

Valódi programozási nyelvek szeletelése során számos olyan probléma merül fel, amelyek megoldása nélkül a szeletelés megvalósíthatatlan. Ilyen például a változók hatókörének, az összetett utasításoknak, a dereferenciáknak vagy az eljárások paraméterkezelésének a problémája, illetve a végrehajtási történet előállítása. Az alábbiakban részletesen kifejtjük, hogy az említettek miért is okoznak, okozhatnak komoly problémát a $d : U$ alapú szeletelés megvalósítása során.

7.1.1. Változók, mutatók és egyéb összetett adatstruktúrák

Az egyszerű algoritmus egyszerű, statikus $d : U$ reprezentációval dolgozik. Ennek az egyik összetevője, hogy minden egyes változónév, ami a $d : U$ reprezentációban megjelenik egyedi, tehát önmagában azonosítja a változót. A valós programozási nyelvekre

viszont ez nem igaz. Valós programozási környezetben előfordul, hogy ugyanaz a változónév a futás során más-más változót jelent. Egyrészt változókat (megfelelően szabályozott módon, például utasításblokkban vagy függvényekben) újra lehet deklarálni, ezáltal az új változó hatókörében a megadott névvel nem a régi, hanem az új változót érjük el. Másrészt ha szintaktikailag ugyanaz is a változó (például egy függvény lokális változója), a program futása során akkor is előfordulnak olyan helyzetek, amikor más-más változónak kell tekintenünk a futás közbeni különböző előfordulásait.

A változók problémájához nagyon hasonló probléma a mutatók (pointerek) és egyéb összetett adattípusok problémája. Egyrészt, a pointer által dereferált memóriaterület a fent említett eseteknél sokkal gyakrabban és a program struktúrájától függetlenül változhat. Másrészt itt az a probléma is fennáll, hogy két mutató ugyanarra a memóriaterületre hivatkozik. Tömbök és struktúrák esetében pedig nyilvánvalóan különbséget kell tennünk a teljes adattípusra és az adattípus egy elemére való hivatkozások között.

Szerencsére a fenti problémák egységesen kezelhetők, amennyiben az algoritmusunkat úgy módosítjuk, hogy az szimbolikus nevek helyett memóriacímekkel azonosítsa a változókat. Ez egyrészt pontosabb mint a szimbolikus nevek, mert akár logikailag hibás kódrészletekben is felfedezheti a jelenlévő technikai függőségeket[1], másrészt megvalósítható, hiszen futás közben a memóriacímeket pontosan ismerjük. Ennek megvalósításához persze többféle technikai megoldásra is szükség van, amiket részletesen a 7.2 fejezetben ismertetünk.

7.1.2. Vezérlési függőségek

Egy strukturált programozási nyelvben a statikus vezérlési függőségek meghatározása nem okoz gondot. De a valós programozási nyelvek legtöbbje tartalmaz nem strukturált vezérlési utasításokat is, mint például C-ben a `break`, `continue` vagy a `goto`.

A vezérlési folyam gráfból ilyen esetekben is építhető függőségi gráf ([2, 43]), a vezérlési függőségeket átültetni az általunk használt $d : U$ reprezentációra pedig már nem jelent gondot. Olyan esetben pedig, amikor egy utasítás statikusan több más utasítástól is kontroll-függ, dinamikusan csak a közülük utoljára végrehajtott elem vezérlési függőségét kell figyelembe venni.

[1] Ilyen lehet például az, ha egy függvény veremben keletkező lokális változóját nem inicializáljuk. Ez ilyenkor az előzőleg – valamely más változó által – „ottfelejtett” értéket fogja használni, ami adatfüggőséget jelent. Ezt szimbolikus számításokkal eléggé nehéz lenne pontosan detektálni.

7.1.3. Eljárások közötti analízis

A szeletelő algoritmusaink a megadott formában intraprocedurálisak, ami azt jelenti, hogy csak eljárásokon belül számítanak szeleteket, ami valós programozási nyelveknél nyilvánvalóan nem elegendő. Eljárások közötti analízis esetén a problémát az okozza, hogy az egyes függvények több helyről hívhatóak, így az egyes futások során más-más függőségek realizálódhatnak bennük illetve rajtuk keresztül. Meg kell tehát oldani a függvényhívásoknál az aktuális adatáramlásból adódó függőségek modellezését.

Ehhez – a predikátumváltozókhoz hasonló módon – virtuális változókat vezetünk be. Egy függvény minden paramétere változóként van kezelve, míg a függvényhívás minden egyes argumentumához szintén hozzárendelünk egy-egy változót. Ezek a változók azután a futás során az aktuális függvényhívásnál össze lesznek kapcsolva, így a függvénybe lépéskor ezeken keresztül ugyanúgy tudunk majd függőségeket számolni, mint bármely más változókon keresztül. A függvények visszatérési értéke hasonlóképpen van kezelve: a függvény minden visszatérési helyén definiálunk egy változót a visszaadott értékhez, amit a függvény hívásának helyén használunk fel.

7.1.4. Többszálúság

Problémát okoz az, ha egy program több szálon fut, hiszen ilyenkor nem egyetlen vezérlési illetve adatfolyamot kell kezelni, hanem egyszerre többet. Ezek ráadásul nem feltétlenül szeparáltak, hanem bizonyos pontokon kapcsolódnak egymáshoz.

A Java implementációnk képes a többszálú Java programok szeletelésére, amennyiben nincs benne szinkronizálatlan adathozzáférésből adódó ütközés. Ennek elve az, hogy a szálindítás, szálváltás és a szál törlésének eseményeit kezelve minden szálon külön szeletelünk. Ezek a szeletelők külön, de nem egymástól függetlenül dolgoznak, és a szeleteket a végén összegezzük, ha szükséges.

7.1.5. Kivételkezelés

A modern nyelvekben a hibakezelés kivételekkel történik. Ez a programok normál vezérlési folyamában található lehetséges végrehajtási útvonalakat jócskán megtöbbszörözi. Ráadásul ez már nem csak eljáráson belüli, hanem eljárások közötti végrehajtási útvonalakat jelent. Némiképpen tehát a nem strukturált vezérlési függőségek és az eljárások közötti analízis problémáit ötvözi.

A Java implementációnkban megoldottuk a kivételkezelés problémáját is, méghozzá eljárásokon belül épített alternatív vezérlési folyam gráfok segítségével, amelyeket kivételdobás esetén használunk a normál gráfok helyett. Ez és a Java-s metóduskezelésünk

együtt képes rendben kezelni a programok kivételes működését.

7.1.6. Nyomkövetési információ

A dinamikus szeletelési algoritmusok implementálásának egyik sarokpontja a végrehajtási nyom kinyerésének implementálása. Ezt különböző nyelvekre sokféleképpen meg lehet oldani. Általános megoldás a forráskód instrumentálása, azaz módosítása olyan utasítások beszúrásával, amik a kód eredeti viselkedését nem változtatják meg, de kiegészítik olyan műveletekkel mint például az utasítássorszám, blokkba belépés, függvény egyértelmű azonosításához szükséges információk vagy a használt memóriacímek kiírása. Hasonló elven megoldható a binárisok instrumentálása is. Olyan esetekben, amikor a futtatás nem natív módon, hanem egy program által vagy felügyeletével történik, lehetőség van ezen programok átalakítására. Ilyen például a Java virtuális gép [55] vagy a GDB [11], amely kiegészíthető úgy, hogy előállítsa a megfelelő EH-t.

A C nyelv esetében mi a forráskód instrumentálása mellett döntöttünk többek között azért, mert rendelkezésünkre állt a CAN [23] statikus elemző eszköz, amelyet kiegészítve elő tudtuk állítani az instrumentált kódot.

7.2. Megoldások C nyelvre

Ebben az alfejezetben a C nyelv szeletelésénél előkerült problémák közül a C nyelv változóinak, mutatóinak és összetett típusainak a kezelését, valamint az ehhez szükséges $d : U$ átalakítást mutatjuk be részletesen.

7.2.1. *d:U* átalakítása

A C nyelv szeleteléséhez az egyik legnagyobb módosítást a változó fogalmának átalakítása jelenti. Eddig változóként a program statikus reprezentációjában megtalálható adattároló elemre hivatkoztunk. Ezt most megváltoztatjuk úgy, hogy a statikusan elérhető elem helyett az ő aktuális dinamikus megvalósulását tekintjük változónak. Ezzel az átalakítással – mint majd látjuk – a változók hatókörének, a mutatóknak, összetett adatszerkezeteknek és a függvényhíváson keresztüli adatáramlásnak a kezelését is meg tudjuk oldani. Ehhez bevezetjük a *dinamikus definíció-használat* (dynamic definition-use) relációt, amit a statikus $d : U$ reprezentációból és különféle, a végrehajtási nyomba rejtett információból (pl. változó címe, mutató értéke) számítunk. Míg a statikus $d : U$ reprezentáció a $d : U$ párokat a program utasításaihoz rendeli, addig virtuálisan az EH minden végrehajtott utasításpéldányához tartozik egy dinamikus $d : U$ pár, ami az ak-

tuális dinamikus függőségeket tartalmazza. A dinamikus $d : U$ előállításának módját részletesen a későbbiekben ismertetjük.

Az előző fejezetben egy utasítás $d : U$ reprezentációját $i.\ d_i : U_i$ -ként adtuk meg. A C nyelvben viszont egyetlen utasítás rengeteg értékadó műveletet tartalmazhat, azaz egyetlen utasítás nem csupán egy változót definiálhat. Ezért egy kicsit átalakítjuk a statikus $d : U$ reprezentációt is. Egy utasításhoz nem egyetlen $(d : U)$ párt, hanem $(d : U)$ párok sorozatát rendeljük, azaz az i-ik utasításhoz rendelt $d : U$ elem:

$$i.\ \langle (d_{i,1} : U_{i,1}),\ (d_{i,2} : U_{i,2}),\ \ldots \rangle\ .$$

Az utasításon belül található minden egyes értékadó művelethez pontosan egy $d : U$ pár tartozik. Az utasítás reprezentációján belül az egyes $d : U$ párok sorrendje fontos, ezt az utasításon belüli részkifejezések kiértékelési sorrendje[2] határozza meg, és így a korábban definiált $d_{i,j}$ változókat a későbbi $U_{i,k}$ halmazok felhasználhatják.

Instrumentálás és végrehajtási nyom

Ahhoz, hogy a statikus $d : U$ reprezentációból elő tudjuk állítani a dinamikus $d : U$ elemeit, az adott akción kívül sok egyéb adatra is szükségünk van, mint például memóriacímek, blokkba belépés vagy függvény elhagyásának érzékelése. Ezekhez az információkhoz csakúgy, mint magához az akciók sorozatához a végrehajtási nyom fogalmának kiterjesztésével jutunk. Azon túl tehát, hogy ez a nyom tartalmazza az eddigi *EH*-t, tartalmaz minden olyan segédinformációt is, melynek segítségével a statikus $d : U$ reprezentációból meghatározható egy-egy utasítás dinamikus $d : U$ reprezentációja. A kiterjesztett végrehajtási nyomra *TRACE*-ként fogunk hivatkozni. A plusz adatokat – ugyanúgy, mint magát az *EH*-t – instrumentálás segítségével kapjuk. Az továbbiakban egyes konstrukciók kezelésénél megadjuk az instrumentáló utasításokat is.

7.2.2. Változók, mutatók

Ahhoz, hogy a mutatókat és a dinamikusan foglalt változókat is kezelni tudjuk, az algoritmus futása során mindenféle (direkt vagy indirekt) változóelérést memóriacímeléréssé konvertálunk. A programban található dereferenciákhoz a statikus $d : U$ reprezentációban egy-egy egyedi szimbolikus nevet társítunk. Nyilvánvaló, hogy a program futása során egy adott változónévhez tartozó cím vagy egy mutató értéke „bármikor” megváltozhat, ezért ezeket az értékeket az adott akcióknak megfelelően folyamatosan

[2] A C nyelvben a részkifejezések kiértékelési sorrendje nem minden esetben rögzített. Ilyenkor a C környezetfüggetlen nyelvtanának elemzési sorrendjére hagyatkozunk.

karbantartjuk. Ezt megtehetjük, hiszen minden egyes akcióban egyértelműen azonosíthatóak a változók címei vagy a mutatók értékei. Így a dinamikus függőség-számítás során már memóriacímekkel, és nem szimbólikus nevekkel dolgozhatunk, ezáltal képesek vagyunk függőségek számítására bármely memóriacímen keresztül függetlenül attól, hogy az adott címet változón vagy mutatón keresztül illetve bármely más módon értük-e el. Tömbök, tömbelemek, struktúrák mezőinek elérése szintén átalakítható memóriacímhivatkozássá, tehát ezek is teljesen hasonló módon kezelhetőek.

A továbbiakban átnézzük a fenti elv megvalósításának módját és lépéseit C nyelvű szeleteléshez.

Változók

A változók kezelésével viszonylag egyszerű dolgunk van: az algoritmus futása közben szimulálni kell a változók életciklusát, vagyis tulajdonképpen a vermet. A blokkba belépés/blokkból kilépés eseményeket (most ideértve a függvényblokkot is) rögzítjük a *TRACE*-ben; ezek azért fontosak, mert a változók életciklusát a blokkok határozzák meg. (A globális változókat mint egy legkülső virtuális blokkban deklarált változókat lehet kezelni.) A *TRACE*-ben szintén rögzítjük a változók címét a deklarációjuk pontján. A fenti háromféle eseményből már felépíthető egy megfelelő adatstruktúra, amiben minden változóhoz rendelünk egy vermet, és deklaráció esetén ebbe beletesszük az új címet, amikor pedig a deklaráló blokkot elhagytuk, akkor kivesszük. A változó aktuális címe mindig a verem tetején található.

Mutatók

Egy változó címe a hatókörén belül nem változik, ha tehát egyszer meghatároztuk, utána akárhányszor felhasználhatjuk. De egy mutató értékét minden egyes felhasználás alkalmával le kell kérdeznünk, hiszen bármikor megváltozhat. Vegyük a következő példát:

```
    int x, *p;
1:  x  = 1;
2:  p  = &x;
3:  *p = 2;
4:  printf("%d\n", x);
```

Látható, hogy az x változó 4. sorban kiírt értékét csak a program 2. és 3. sorai befolyásolják, az első nem. Statikusan az ilyen jellegű függőségek felderítése majdnem lehetetlen. A legtöbb statikus algoritmus vagy az egész programot a szelet részeként

jelölné, vagy a 2. és 3. sorokat hagyná ki. A fenti példához mi statikusan a következő $d : U$ reprezentációt rendeljük:

$i.$	$d_i : U_i$
1.	$x : \emptyset$
2.	$p : \emptyset$
3.	$PTR1 : \{p\}$
4.	$OUT4 : \{x\}$

ahol *PTR1* jelöli, hogy pointer dereferencián keresztüli memóriaeléréssel van dolgunk. Természetesen az elért memóriacím erősen függ a p értékétől. Dinamikus információt használva a $d : U$ „változói" memóriacímekké konvertálhatók, azaz a dinamikus $d : U$ reprezentáció x és p változók helyett a `$01` és `$02` memóriacímeket használja. Dinamikusan *PTR1* értéke szintén ismert. A fentiek alapján a végrehajtási nyom és a hozzá csatolt információk segítségével az alábbi dinamikus $d : U$-t tudjuk összeállítani az algoritmus futása közben:

akció	$d_i : U_i$	4^4 hátramutató szelete
1^1	`$01` : $\emptyset$	$\emptyset$
2^2	`$02` : $\emptyset$	$\emptyset$
3^3	`$01` : {`$02`}	$\{2\}$
4^4	*OUT4* : {`$01`}	$\{2, 3\}$

A különféle dinamikus információk kinyerését a program instrumentálásával valósítjuk meg. A fenti kódrészleten a mutatók kezelése miatt elvégzett instrumentálás[3] a következő kódrészletet eredményezi:

```
    int x, *p;
    remember("x", &x);
    remember("p", &p);
1:  x = 1;
2:  p = &x;
3:  *dump("PTR1", p) = 2;
4:  printf("%d\n", x);
```

A `remember` függvény végzi a változók címeinek *TRACE*-be íratását. A kiírt értékeket a dinamikus $d : U$ előállításához fogjuk felhasználni. A `dump` függvény (jelen esetben)

[3]Az instrumentált kódrészletekben csak az adott feladat szempontjából lényeges instrumentációt tüntetjük fel. A teljes instrumentálás ezeknél jóval bonyolultabb és olvashatatlanabb kódot eredményez.

a mutató értékeinek kiírását végzi úgy, hogy a mutatóval vissza is tér, így az használható mint *l-value*. A statikus $d : U$ megfelelő sora tartalmazza a *PTR1* elemet, ennek a memóriacímmé történő feloldását teszi lehetővé a kiíratás.

Tömbök

A C nyelvben – szerencsénkre – a tömbök és a mutatók gyakorlatilag ugyanazok, és a konverzió nyelvi szinten is támogatott, tehát a tömbök kezelése meglehetősen egyszerű. Egy `t` tömb i-ik eleme C-ben tömbjelölést használva `t[i]`-ként írható fel, de semmi akadálya nincs annak, hogy ezt `*(t+i)` alakban mutatóként hivatkozzuk. Ez a C nyelv egyik adottsága. A megoldás tehát kézenfekvő, minden tömbelem hivatkozást alakítsunk át, és kezeljünk mutatóként. Lássuk a következő példát:

```
    int t[5];
1:  t[0] = 1;
2:  *(t+1) = 2;
3:  t[2] = 2 * t[1];
4:  printf("%d\n", t[2]);
```

Ennek a programrészletnek a statikus $d : U$ ábrázolása a következő lesz:

$i.$	$d_i : U_i$
1.	$PTR1 : \emptyset$
2.	$PTR2 : \emptyset$
3.	$PTR3 : PTR4$
4.	$OUT4 : PTR5$

A tömbelem-hivatkozásokat – amint azt fentebb említettük – pointer-dereferenciákká konvertáltuk. A forráskód minden egyes mutató-hivatkozásának és tömbelem-elérésének egyedi azonosítója lesz a statikus $d : U$-ban. Az instrumentált kód[3] a következő:

```
    int t[5];
    remember("t", t);
1:  *dump("PTR1", &t[0]) = 1;
2:  *dump("PTR2", t+1) = 2;
3:  *dump("PTR3", &t[2]) = 2 * (*dump("PTR4", &t[1]));
4:  printf("%d\n", *dump("PTR5", &t[2]));
```

Ha feltételezzük, hogy a `t` tömb a futás során a memóriában a `$10` címre került, akkor a szeletszámítás közben az alábbi dinamikus $d : U$-val dolgozhatunk:

akció	$d_i : U_i$	4^4 hátramutató szelete
1^1	$\$10 : \emptyset$	$\emptyset$
2^2	$\$11 : \emptyset$	$\emptyset$
3^3	$\$12 : \{\$11\}$	$\{2\}$
4^4	$OUT4 : \{\$12\}$	$\{2, 3\}$

Struktúrák

A struktúrák alapvetően a skaláris változókhoz hasonlítanak: a struktúrák mezőinek struktúrán belüli címe (offszet) statikus időben meghatározható. De ebből az információból dinamikus címeket előállítani meglehetősen komplikált lenne például akkor, amikor magát a struktúrákat eleve pointeren keresztül érjük el. Ezért a struktúrák mezőit is inkább eleve mutatókként kezeljük. Ezáltal a struktúrák kezelését is visszavezettük a mutatók kezelésére. Lássuk az alábbi példát:

```
    struct s {
        int a;
        int b;
    };
    struct s x,y;
1:  x.a = 1;
2:  x.b = 2;
3:  y = x;
4:  printf("%d\n", y.b);
```

Itt három mezőelérés van: `x.a` és `x.b` az első illetve második sorban, és `y.b` a negyedikben. Ezeket rendre *PTR1*, *PTR2* és *PTR3* pointerekké konvertáljuk. A teljes `x` és `y` struktúrákat nem konvertáljuk mutatókká, azokat skaláris változóként kezeljük. Így a programrészlet $d : U$ reprezentációja:

$i.$	$d_i : U_i$
1.	$PTR1 : \emptyset$
2.	$PTR2 : \emptyset$
3.	$y : \{x\}$
4.	$OUT4 : \{PTR3\}$

Most a példánk erejéig tegyük fel, hogy a struktúráknak és mezőiknek az alábbi címek lettek futás közben kiosztva:

elem	cím
x	$01
x.a	$01
x.b	$02
y	$03
y.a	$03
y.b	$04

Amint az látható, a cím önmagában nem azonosítja pontosan a változót. Bár `x` és `x.a` címe megegyezik, ezek mégis különböző változók. Míg `x.a` egyetlen memóriacellában elfér, addig `x` két cellát foglal: `$01`-et és `$02`-t. Van még egy ok a struktúra méretének feljegyzésére: a 3. sorban található `y = x` kifejezés. Ha a struktúra méretét figyelmen kívül hagyjuk, akkor az `x.b` és `y.b` közötti reláció felfedezhetetlen. Az instrumentálás során beszúrt `remember` függvénynek van tehát egy további paramétere, a változó mérete. A struktúrák mezőinek címeit a mutatóknál is használt `dump` függvénnyel írjuk ki. Az instrumentált kód:

```
    struct s {
        int a;
        int b;
    };
    struct s x, y;
    remember("x", &x, sizeof(x));
    remember("y", &y, sizeof(y));
1:  *dump(&x.a) = 1;
2:  *dump(&x.b) = 2;
3:  y = x;
4:  printf("%d\n", *dump(&y.b));
```

A statikus $d : U$ reprezentáció és a *TRACE*-be írt dinamikus információk alapján a következő dinamikus $d : U$-t lehet összeállítani:

akció	$d_i : U_i$	4^4 hátramutató szelete
1^1	$01 : $\emptyset$	$\emptyset$
2^2	$02 : $\emptyset$	$\emptyset$
3^3	$03 : {$01}	$\{1\}$
	$04 : {$02}	$\{2\}$
4^4	*OUT4* : {$04}	$\{2, 3\}$

A 3. lépésben két memóriacímet is definiálunk, mindkettőt a saját függőségével. Amikor az `y = x` kifejezést kiértékeljük, az `x` struktúra teljes tartalmát átmásoljuk az `y`-ba. Ez azt jelenti, hogy a `$01`-en kezdődő két cella értéke lesz átmásolva a `$03` címen kezdődő helyre. Az algoritmus felismeri ezt és a statikus $d : U$ egyetlen $y : \{x\}$ eleméből létrehozza a két dinamikus függőséget.

Memóriaterületek méretei

A struktúra esetéhez hasonlóan minden változó és pointer által mutatott terület méretét ismerni kell. Nyilvánvaló, hogy egy mutató bármilyen struktúrára mutathat, vagy hogy struktúrákból is készíthetünk tömböt. Így a `dump` függvénynek valójában szintén van egy plusz paramétere, ami a pointer által mutatott típus mérete. Ilyen módon az algoritmus elő tudja állítani a megfelelő dinamikus $d : U$-t, és az összes használt vagy definiált memóriacellát megtalálja.

Végrehajtás alatti azonos memóriacímek

A program végrehajtása során a memóriában rengeteg (implicit vagy explicit) helyfoglalás illetve memória felszabadítás történik. Előfordulhat, hogy egy ilyen felszabadítás után egy másik változó megkapja ugyanazt a címet. Ez azonban nem okoz problémát, hiszen ha a változókat megfelelő módon inicializáltuk, akkor a függőség csak az inicializáló utasításig vezethető vissza. Ha viszont a változó nem lesz inicializálva, a függőséget az előző értéket kiszámító utasításig vissza lehet vezetni. De a program valóban így fog működni, hiszen akkor a memóriában is pontosan ennek a számításnak az eredménye marad benne, tehát a kapcsolat a régi és az új változó között – ha hibás működés következtében is, de – fennáll.

7.2.3. Módosított algoritmus C nyelvre

Ebben az alfejezetben bemutatjuk, hogy hogyan működik a C nyelvű programok szeletelése, és konkrétan a hátramutató szeleteket előrehaladó *EH* feldolgozással számító globális algoritmus.

Először is, a mutatók kezelésének szükségessége a szeletelési kritérium értelmezésének enyhe bővítését vonja maga után. Ez azt jelenti, hogy ha például egy `*p` pointer dereferencia dinamikus szeletét szeretnénk meghatározni, akkor tulajdonképpen egy memóriacím dinamikus függőségeit keressük (és nem egy változóét, mint az eredeti definíciónkban).

A C programokat szeletelő algoritmusunk a következő főbb lépésekből áll.

- Először elemzéssel előállítjuk a program statikus $d : U$ reprezentációját, és instrumentáljuk is a programot hogy előállítsa a szükséges futási információkat.
- Azután az instrumentált programot lefordítjuk és futtatjuk, aminek eredménye a $TRACE$, az EH-t és egyéb dinamikus információkat (például memóriacímeket, blokkba belépést) tartalmazó adatsor.
- Végül a dinamikus szeletelési algoritmust futtatjuk a statikus $d : U$ és a dinamikus $TRACE$ segítségével, aminek az eredményei a kritériumoknak megfelelő szeletek.

Egy program statikus $d : U$ reprezentációjában található „változóknak" többféle jelentésük lehet, ezek a következők:

Skaláris változók Ezek a „reguláris" lokális vagy globális változók, amiknek a deklarációtól kezdve a hatókörükön belül állandó címük van.

Predikátum változók Az ilyen változókat Pn-nel jelöljük, ahol n a megfelelő predikátum utasítás sorszáma, ahogyan azt korábban említettük.

Kimeneti változók A kimeneti változók jelölése $OUTn$, ahol n szintén utasítás-sorszámot jelöl. Definíció szerint a kimeneti változó egy olyan ál-változó, amit azokra a helyekre generálunk, ahol az U halmazt használjuk ugyan, de semmilyen más változó nem kap értéket U-ból. Mint például az eldobott visszatérési értékű függvényhívási helyek, mellékhatás nélküli egyszerű kifejezés-utasítások, és az egyszerűség kedvéért a C néhány kimeneti utasítása (pl. `printf`).

Dereferencia változók Jelölésük $PTRn$, ahol n egy globális számláló az összes dereferenciához. Ott használunk dereferencia változókat, ahol egy memóriacímet használunk vagy annak adunk értéket mutatón (vagy az előző fejezetben tárgyalt azzal ekvivalensként kezelt konstrukción) keresztül.

Függvényhívás argumentum változó Jelölésük $ARG(f, n)$, ahol f a függvény neve n pedig az argumentum sorszáma. Egy argumentum változót a függvényhívás helyén *definiálunk* és a függvény belépési pontján *használunk*.

Függvényhívás visszatérési változó Ezeket $RET(f)$-fel jelöljük, ahol f a függvény neve. Egy ilyen változót a függvény kilépési pontján *definiálunk* és a függvény hívásának helyén *használunk*.

A 7.1. ábrán egy C nyelvű példaprogramot és a statikus $d : U$ reprezentációját láthatjuk a fenti jelöléseknek megfelelően.

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`int a, b;`	
1.	`int f(int x, int y) {`	$x : \{ARG(f,1)\}$, $y : \{ARG(f,2)\}$
2.	`  a += x;`	$a : \{a, x\}$
3.	`  b += y;`	$b : \{b, y\}$
4.	`  return x+2;`	$RET(f) : \{x\}$
	`}`	
5.	`int g(int y) {`	$y : \{ARG(g,1)\}$
6.	`  a += y;`	$a : \{a, y\}$
7.	`  return y+1;`	$RET(g) : \{y\}$
	`}`	
	`void main() {`	
	`  int s, *p;`	
8.	`  s = 0;`	$s : \emptyset$
9.	`  scanf("%d", &a);`	$a : \emptyset$
10.	`  scanf("%d", &b);`	$b : \emptyset$
11.	`  p = &b;`	$p : \emptyset$
12.	`  while (*p < 10) {`	$P12 : \{p, PTR1\}$
13.	`  s += f(3,4);`	$ARG(f,1) : \{P12\}$, $ARG(f,2) : \{P12\}$, $s : \{s, RET(f), P12\}$
14.	`  s += g(3);`	$ARG(g,1) : \{P12\}$, $s : \{s, RET(g), P12\}$
	`  }`	
15.	`  printf("%d", *p);`	$OUT15 : \{PTR2\}$
16.	`  printf("%d", s);`	$OUT16 : \{s\}$
	`}`	

7.1. ábra. Példa: egy C példaprogram és (statikus) $d : U$ reprezentációja

Miután meghatároztuk a program $d : U$ reprezentációját és egy adott végrehajtáshoz a *TRACE*-t, kiszámolhatjuk az adott kritériumhoz tartozó szeleteket. A *TRACE* feldolgozását az elején kezdjük, és a benne található különböző típusú elemek esetén más-más műveleteket végzünk. Ha az aktuális elem egy adminisztratív elem, akkor elvégezzük a megfelelő műveleteket a belső reprezentáción (mint például egy verem karbantartása a hivatkozható változók számára a hatókörök karbantartása miatt). Ha az elem egy *EH* elem akkor feldolgozzuk a megfelelő $i.\ \langle d : U \rangle$ sorozatot a következőképpen. A sorozat minden eleméhez meghatározzuk a megfelelő d' és $u'_k \in U$ dinamikus függőségeket, kiszámoljuk a $DynSlice(d')$ halmazt és $LS(d')$-t az alábbiak szerint:

$$DynSlice(d') = \bigcup_{u'_k \in U'} \Big(DynSlice(u'_k) \cup \{LS(u'_k)\} \Big),$$

$$LS(d') = i.$$

A korábban leírtak alapján ez a számítási sorrend kötött. Az említett d' és u'_k értékek a

megfelelő d és u_k értékekből számolódnak az alábbiak szerint.

- Ha d *skaláris változó* akkor d' az ő memóriacíme lesz, ami a *TRACE* eddig feldolgozott részéből megállapítható.
- Ha d egy *dereferencia változó* akkor d' a megfelelő mutató értéke lesz, ami szintén a *TRACE* alapján határozható meg.
- Ha d *predikátum változó* akkor a Pn változót kiegészítjük egy „mélységgel" ami a függvényhívási lánc mélységének felel meg, és ezt $Pn(k)$-val jelöljük. Erre az esetleges rekurziók miatt van szükség, amikor ugyanannak a változónak több előfordulását külön változóknak kell tekintenünk.
- Minden más esetben d és d' megegyezik.

Függvényhívás esetén az aktuális $d : U$ sorozat nem dolgozható fel egyben. Mi a függvényhívásokat és visszatéréseket összesen négy virtuális lépésben kezeljük. Egyrészt a hívási helyen lesz egy paraméter-átadási lépés, amikor az aktuális paraméterek bemásolódnak a megfelelő virtuális argumentum változókba, majd a függvényhívási lépés, amikor a feltöltött argumentumváltozók értékei a hívott függvényben bemásolódnak a megfelelő lokális változókba (C-ben a függvény paraméterei gyakorlatilag a hívás helyén inicializált lokális változók). Másrészt visszatéréskor még a függvényben az aktuális érték bemásolódik egy visszatérési változóba, amin keresztül azután a hívási helyen felhasználjuk. Így viszont az i. hívási utasításhoz rendelt $d : U$ sorozat feldolgozása közben, az argumentumok átadása és a visszatérési érték átvétele között fel kell dolgozni a függvényhez tartozó utasításokat is. Ilyenkor egy veremben eltároljuk, hogy éppen hol járunk az i. utasítás feldolgozásában, és visszatérés után innen folytatjuk a feldolgozást.

Ezek után már formalizálni tudjuk a C programok szeletelésére alkalmas globális szeletelő algoritmust, amit a 7.2. ábrán láthatunk. Az algoritmus az egyes memóriacímek utolsó definíciójára számolja ki a szeleteket.

Az algoritmus működését a 7.1. ábrán látható példaprogramon keresztül illusztráljuk az $\mathbf{x} = \langle$`a=2`, `b=6`$\rangle$ bemeneti adatokon. Ezen az inputon a végrehajtási történet $EH = \langle 8, 9, 10, 11, 12, 13/1, 2, 3, 4/13, 14/5, 6, 7/14, 12, 15, 16\rangle$. A kettős 13/1, 4/13, 14/5 és 7/14 elemek felelnek a függvényargumentum és visszatérési érték átadásáért, ahogyan azt korábban leírtuk. A paraméter-átadás felfogható két utasítás kombinációjaként egyetlen akcióban, hiszen először a hívó utasítása elhelyezi az átadott értéket a veremben, amit azután a hívott függvény utasítása kiolvas. A visszatérési érték átadása hasonló módon interpretálható. Az algoritmus futása során kiszámított értékek a 7.3. ábrán láthatóak. A `$xxxxxxx` alakú értékek a *TRACE* által szolgáltatott memóriacímek.

program DynamicSliceForC($P, \mathbf{x}$)

input: P : a program
$\mathbf{x}$: a program input

output: hátramutató szelet az összes CB kritériumhoz

```
    begin
1:      Inicializáljuk az LS és DynSlice struktúrákat
2:      Építsük fel a d : U reprezentációt
3:      Read(TRACE)
4:      actualDUItem := nil
5:      for actualTraceItem := First(TRACE) to Last(TRACE) do
6:          case actualTraceItem of
7:          'function begin' esemény:
8:              push(actualDUItem)
9:              actualDUItem := nil
10:         'function end' esemény:
11:             pop(actualDUItem)
12:         i^j ∈ EH akció:
13:             actualDUItem := i.⟨d : U⟩ első eleme
14:         egyéb:
15:             Oldjuk fel a feloldatlan memóriahivatkozásokat
                    az aktuális d : U elemben
            endcase
16:         while actualDUItem feldolgozható* do
17:             actualDUItem alapján határozzuk meg d'-t és U'-t
18:             DynSlice(d') := ⋃_{u'_k ∈ U'} (DynSlice(u'_k) ∪ {LS(u'_k)})
19:             LS(d') := i
20:             actualDUItem := i.⟨d : U⟩ következő eleme
            endwhile
        endfor
21:     Output(LS ∪ DynSlice) minden memóriacím utolsó definíciójára
    end
```

*Ez igaz, ha:
(1) a statikus $d : U$ alapján az aktuális pozícióban nincs függvényhívás **és**
(2) $actualDUItem \neq nil$ **és**
(3) $actualDUItem$ nem tartalmaz feloldatlan memóriahivatkozásokat

7.2. ábra. DynamicSliceForC algoritmus: C programok szeletelésére

A $CB_{14} = (\langle \texttt{a=2}, \texttt{b=6} \rangle, 15^{14}, \{\texttt{*p}\})$ kritériumhoz tartozó szelet $DynSlice(OUT15)$ és $\{LS(OUT15)\}$ uniója. Az eredmény a 7.4. ábra megfelelő oszlopából is kiolvasható, ahol is a megjelölt sorok tartoznak a szelethez. Mint látható, a szelet azokat az utasításokat tartalmazza, amelyek hatással vannak a `p` által mutatott `$4347828` memóriacímre (ami valójában a `b` skaláris változó memóriacíme).

Érdekességképpen, a $CB_{15} = (\langle \texttt{a=2}, \texttt{b=6} \rangle, 16^{15}, \{\texttt{s}\})$ dinamikus kritériumhoz tartozó szelet (a 7.4. ábrán az utolsó oszlop) csak azokat az utasításokat tartalmazza, amik a

akció	d'	U'	$DynSlice(d')$	$LS(d')$
8^1	$6684144	$\emptyset$	$\emptyset$	8
9^2	$4347824	$\emptyset$	$\emptyset$	9
10^3	$4347828	$\emptyset$	$\emptyset$	10
11^4	$6684148	$\emptyset$	$\emptyset$	11
12^5	$P12(1)$	{$6684148, $4347828}	{10,11}	12
13^6	$ARG(f,1)$	$\{P12(1)\}$	{10,11,12}	13
13^6	$ARG(f,2)$	$\{P12(1)\}$	{10,11,12}	13
1^6	$6684060	$\{ARG(f,1)\}$	{10,11,12,13}	1
1^6	$6684064	$\{ARG(f,2)\}$	{10,11,12,13}	1
2^7	$4347824	{$4347824, $6684060}	{1,9,10,11,12,13}	2
3^8	$4347828	{$4347828, $6684064}	{1,10,11,12,13}	3
4^9	$RET(f)$	{$6684060}	{1,10,11,12,13}	4
13^9	$6684144	{$6684144, $RET(f), P12(1)$}	{1,4,8,10,11,12,13}	13
14^{10}	$ARG(g,1)$	$\{P12(1)\}$	{10,11,12}	14
5^{10}	$6684064	$\{ARG(g,1)\}$	{10,11,12,14}	5
6^{11}	$4347824	{$4347824, $6684064}	{1,2,5,9,10,11,12,13,14}	6
7^{12}	$RET(g)$	{$6684064}	{5,10,11,12,14}	7
14^{12}	$6684144	{$6684144, $RET(g), P12(1)$}	{1,4,5,7,8,10,11,12,13,14}	14
12^{13}	$P12(1)$	{$6684148, $4347828}	{1,3,10,11,12,13}	12
15^{14}	$OUT15$	{$4347828}	{1,3,10,11,12,13}	15
16^{15}	$OUT16$	{$6684144}	{1,4,5,7,8,10,11,12,13,14}	16

7.3. ábra. Példa: DynamicSliceForC algoritmus egy lefutása

skaláris `s` változóra hatással vannak. A két globális változó értékét befolyásoló utasítások nem tartoznak ide, mivel `s` csak az `f` és `g` függvények visszatérési értékeit használja fel, amik konstans értékek.

7.3. Java implementáció

Ebben a fejezetben a szeletelő algoritmusaink Java implementációjának jellemzőit mutatjuk be. A *Jadys* nevű Java szeletelő eszköz szerzőtársam, Szegedi Attila munkája [55]. Ez eredetileg a hátramutató szeleteket számító praktikus algoritmust valósította meg, ezt egészítettük ki az előremutató szeleteket számító párhuzamos algoritmus egy gyengített változatával, amiben a kritériumunk nem a dinamikus kritériumokra jellemző akció-, hanem a statikus kritériumra jellemző utasítás-meghatározást tartalmaz.

7.3.1. A keretrendszer

Az instrumentálás elengedhetetlen a végrehajtási nyom kinyeréséhez, ami alapján a szeleteket számoljuk. Java esetén lehetőség van a keretrendszer, azaz a Java virtuális gép (JVM) instrumentálására is, mi erre a célra a nyílt forráskódú *JamVM*-et [32] használtuk. Az ilyen instrumentálásnak megvannak az előnyei a kód (forráskód vagy bájtkód)

		*p	s
	`#include <stdio.h>`		
	`int a, b;`		
1.	`int f(int x, int y) {`	•	•
2.	`  a += x;`		
3.	`  b += y;`	•	
4.	`  return x+2;`		•
	`}`		
5.	`int g(int y) {`		•
6.	`  a += y;`		
7.	`  return y+1;`		•
	`}`		
	`void main() {`		
	`  int s, *p;`		
8.	`  s = 0;`		•
9.	`  scanf("%d", &a);`		
10.	`  scanf("%d", &b);`	•	•
11.	`  p = &b;`	•	•
12.	`  while (*p < 10) {`	•	•
13.	`  s += f(3,4);`	•	•
14.	`  s += g(3);`		•
	`  }`		
15.	`  printf("%d", *p);`	•	
16.	`  printf("%d", s);`		•
	`}`		

7.4. ábra. Példa: az $\langle a = 2, b = 6 \rangle$ inputra számított szeletek

instrumentálásával szemben. Egyrészt így olyan kódrészeken keresztül is tudunk függőségeket számolni, amiknek a forráskódja nem ismert (például Java könyvtári osztályok, harmadik fél által adott vagy helyben generált kódok), másrészt az instrumentálást csak véges mennyiségű kódon kell végrehajtani, a virtuális gép kódján. Harmadrészt így bármilyen forrásnyelvű programot szeletelhetünk, ami Java bájtkódra lefordítható.

A keretrendszer működése és felépítése a következő. Az instrumentált JVM „futtatja” az osztályokat és előállítja a *TRACE*-t. Ebből azután a szeletelő a nyom előrehaladó feldolgozásával előállítja a hátra- és előremutató szeleteket. Ehhez először egy szálkezelő szétválogatja a szálakat. Minden egyes szálhoz külön szeletszámító tartozik, ezeket a szálkezelő tartja karban (indítja el, állítja le). Van még egy statikus analizátor a rendszerben, ami minden egyes osztályt egyszer (a betöltéskor) leelemez, és előállítja a statikus reprezentációját, amit majd a szeletszámításhoz használunk.

7.3.2. Statikus elemző

A metódusok statikus elemzése három kulcsfontosságú műveletből tevődik össze: bájtkód visszavezetése forrásnyelvi kifejezésekre, a keresési kör szűkítése, vezérlési függőségek számítása.

A szeletelő egy fontos tulajdonsága az a képesség, hogy a generált szeleteket forráskód szintű szimbolikus kritériumokhoz tudja rendelni. A statikus elemző nem számít arra, hogy az osztályok forrás- vagy bájtkódja külön java vagy class fájl formájában elérhető. Ehelyett a *TRACE*-en keresztül kapja meg az osztályok bájtkódját. Ezáltal akár dinamikusan generált kódokat is képes elemezni.

Amikor a szeletelő beolvassa az osztály bájtkódját, eltárolja, így később igény szerint tudja elemezni az osztály metódusait azok első hívásakor. A statikus elemzés része egy egyértelmű szimbolikus azonosító és egy forráskód sorszám hozzárendelése minden egyes bájtkód utasításhoz. Utóbbit az osztályhoz rendelt sorszám-információk alapján teszi meg.

A fenti folyamat eredményeként minden nem-ugró bájtkód utasítás egy statikus szeletelési kritérium lesz. Ebből viszont nagyon sok lehet, így szükség van ezek számának szűkítésére. Az osztályokat reguláris kifejezésekkel tudjuk név alapján szűrni (kihagyni vagy éppen kizárólagosan bevenni a szeletelendő elemek közé).

A vezérlési folyam függőségeinek kezeléséhez minden végrehajtott Java metóduson el kell végezni egy statikus elemzést. A metódus kódját alapblokkokra bontjuk úgy, hogy minden kivételdobó és -kezelő utasításra ugró utasításként illetve ugrás céljaként tekintünk. Ezek után az alapblokkok fölé két vezérlési folyam gráfot húzunk, egyet a normális vezérlési folyamnak, egyet pedig a kivételdobásból adódó lehetséges vezérlési utakkal kiegészítve.

7.3.3. Dinamikus szeletek számítása

A hátramutató szeletek számítása a következőképpen működik. A szeletszámítást a *TRACE* eseményei vezérlik. A szálkezelő folyamatosan olvassa a *TRACE* eseményeit, és ezek nagy többségét változatlan formában továbbítja az éppen aktív szeletszámító felé.

A szeletelési algoritmus adatszerkezeteinek, például az aktuális függőségi halmazoknak a karbantartása is egyfajta eseményvezérelt formában történik. Ez alatt azt értjük, hogy szeletszámítás közben minden adatmanipuláló utasítás maga írja át a művelet céloperandusának megfelelő függőségi halmazokat. Ha az aktuális utasítás része a megfigyelt kódnak, akkor bekerül a megfelelő függőségi halmazba, amibe az aktuális vezérlési

függőségeket is belepakoljuk. A szeletelő az eljáráson belüli aktuális vezérlési függőségeket elágazó utasítások esetén vagy a blokk postdominátorának elérésekor frissíti. Az eljárásközi vezérlési függőségek számításakor az aktív függőségek egy metódusba lépésig vagy a metódus elhagyásáig maradnak érvényben. Mivel Javaban minden metódushívás potenciálisan kivételt dobhat, így alapblokk-határnak tekintjük őket. Egy meghívott metódusban fellépő függőségeket csak a blokk egy postominátorának elérésekor dobjuk el, például normál (nem kivételes) visszatérés esetén. Egyéb esetben a vezérlési függőségek a hívó kivételkezelő blokkjának végéig érvényben maradnak.

Végül, ha az aktuális utasítás egy szeletelési kritérium része, akkor a függőségi halmaza (ami egy dinamikus szeletet határoz meg) tartalmát hozzáadjuk az utasítás eddig számolt és nyilvántartott szeletéhez. Így egy statikus kritériumra számolt kvázi-dinamikus szeletet kapunk.

7.3.4. Uniós szeletek és lefedettség számítása

Miután egy adott program több lefutására is előállítottuk a kvázi-dinamikus szeleteket, a szeletszámító képes ezekből uniós szeleteket összeállítani. Feltételezve, hogy minden végrehajtás ugyanazt a kódbázist használta, az azonos kritériumok a szimbolikus nevek alapján beazonosíthatóak, így az egyes kritériumokhoz tartozó uniós szeletek kiszámítása egyszerű feladat.

Az a mód, amivel a szeletszámítónk a szeletelt program futását emulálja elegendő a precíz utasításszintű lefedettségszámításhoz. A különböző végrehajtásokhoz tartozó lefedettségi adatokat azután persze ugyanúgy összegezni lehet, mint a szeleteket.

7.4. Összefoglalás

Ebben a fejezetben ismertettük, hogy a szeletelő algoritmusainknak milyen konkrét implementációi léteznek. Ismertettük továbbá a konkrét nyelvre történő megvalósítás problémáit, a C nyelvű implementációhoz megoldást adtunk a változók és mutatók kezelésére, bemutattuk a C nyelvre specializált teljes algoritmust, valamint ismertettük a Java nyelvű implementációt.

8. fejezet

d:U alapú szeletelő algoritmusok kiértékelése

Ebben a fejezetben kiértékeljük a szeletelő algoritmusainkat. Egyrészt a 8.1. alfejezetben megadjuk az algoritmusok elméleti komplexitását, és összehasonlítjuk a DDG alapú algoritmuséval, másrészt a a 8.2. alfejezetben az algoritmusok implementációjával kapcsolatos mérések eredményeit közöljük. Az algoritmusok elméleti kiértékelésében a négy új algoritmust tekintve a saját munkám volt meghatározó, a gyakorlati kiértékelések szerzőtársaimmal közös eredmények.

8.1. Szeletelő algoritmusok komplexitásai

Ebben a fejezetben a 6. fejezetben ismertetett hat algoritmus idő- és tárigény szerinti komplexitását vesszük sorba, és összehasonlítjuk a DDG alapú szeletelési algoritmus idő- és tárigényével.

Amikor az idő és tárigényt taglaljuk, kifejezetten az algoritmusok lényegi részére koncentrálunk. Ebbe nem értjük bele a végrehajtási nyom kezdeti beolvasását és tárolását, hiszen ezt minden algoritmusnak meg kell tennie. Ezt a műveletet tekintve egyébként az *EH*-t visszafelé feldolgozó algoritmusoknak nyilvánvaló hátrányuk van, hiszen míg előrefelé feldolgozásnál az *EH* tárolása kiküszöbölhető, addig visszafelé feldolgozásnál az *EH* tárolását nem úszhatjuk meg. (Egy kivétel lesz, az igényvezérelt hátrafelé mutató algoritmus, ahol az *EH* speciális formájának, az *EHT*-nak a felépítését azért számításba vesszük.) A program statikus reprezentációjának felépítését és tárolását – ami a vezérlési függőségek kiszámítását is magában foglalja – szintén nem számoljuk. Mivel itt az elméleti algoritmusok komplexitását taglaljuk, a konkrét megvalósításokhoz szükséges módosítások hatásait is kihagyjuk.

Algoritmus	idő maximum	idő átlag
Igényvezérelt hátramutató	$J \cdot V \cdot \log(J)$	$J + DEP \cdot \log(J)$
Igényvezérelt előremutató	$J \cdot V$	J
Praktikus hátramutató	$J \cdot I \cdot V \cdot \log(I)$	$J \cdot DS \cdot \log(DS)$
Praktikus előremutató	$J \cdot I \cdot V \cdot \log(I)$	$J \cdot DS \cdot \log(DS)$
Párhuzamos hátramutató	$J^2 \cdot (\log(I) + V \cdot \log(J))$	$J \cdot DEP \cdot \log(DS \cdot DEP)$
Párhuzamos előremutató	$J^2 \cdot (\log(I) + V \cdot \log(J))$	$J \cdot DEP \cdot \log(DS \cdot DEP)$
Egy DDG szelet	$J \cdot V$	DEP
DDG építése és egy szelet	$J \cdot V$	$J + DEP$
Összes szelet DDG-vel	$J^2 \cdot V$	$J \cdot DEP$

8.1. táblázat. Dinamikus szeletelő algoritmusok számítási komplexitása

Algoritmus	tárhely maximum	tárhely átlag
Igényvezérelt hátramutató	J	J
Igényvezérelt előremutató	V	V^{DEF}
Praktikus hátramutató	$V \cdot I$	$V^{DEF} \cdot DS$
Praktikus előremutató	$V \cdot I$	$V^{DEF} \cdot DS$
Párhuzamos hátramutató	$J \cdot (I + V)$	$J \cdot DS + V^{DEF} \cdot DEP$
Párhuzamos előremutató	$J \cdot (I + V)$	$J \cdot DS + V^{DEF} \cdot DEP$
DDG	$J \cdot V$	J

8.2. táblázat. Dinamikus szeletelő algoritmusok tárigény komplexitása

Az alábbiakban a következő jelöléseket fogjuk használni. A végrehajtási történet hossza J, a program utasításainak darabszáma pedig I. A program összes változójának számát V, a futás során valóban definiált változóinak számát V^{DEF} jelöli. DS lesz az átlagos szeletméret, míg DEP-pel fogjuk jelölni a DDG egy pontjából egy adott irányban elérhető pontok átlagos számát. Amíg tehát DS az I-vel áll kapcsolatban, addig DEP a J-vel.

A hat $d : U$ alapú és a DDG alapú algoritmus legrosszabb és átlagos idő- és tárigényére vonatkozó számításainkat a 8.1. és a 8.2. táblázatokban foglaltuk össze. A komplexitások összehasonlításánál nem szabad elfelejteni, hogy az igényvezérelt algoritmusoknál feltüntetett értékek egyetlen szelet kiszámítására vonatkoznak, míg a praktikus és párhuzamos algoritmusoknál az összes lehetséges kritériumra kiszámoljuk a szeleteket a megadott költségekkel. A könnyebb összehasonlíthatóság kedvéért a DDG alapú algoritmus időigényét többféle módon is megadtuk.

8.1.1. A DDG alapú algoritmus komplexitása

Hogy a saját algoritmusainkat összehasonlíthassuk a DDG algoritmussal, először ez utóbbi idő- és tárigényét adjuk meg.

A DDG alapú algoritmus tárigényét többféleképpen adjuk meg attól függően, hogy csak egy szelet vagy az összeset ki kell-e számítanunk, illetve hogy a DDG gráf előállítását külön figyelembe vesszük-e vagy sem. Mivel az összes szelet kiszámításánál a gráf felépítése a teljes komplexitást nem növeli meg, a 8.1. táblázatban is látható módon három variációt különböztetünk meg. A különböző értékekre azért van szükség, hogy az összehasonlítás pontosabb legyen. A DDG gráf felépítésének számítási komplexitása legrosszabb esetben $O(J \cdot V)$, hiszen a végrehajtási nyom minden eleméhez tartozik egy külön gráfpont, amiből maximum V (adat)függőségi (és egy vezérlési függőségi) élet kell bekötni. Ez az érték átlagos esetben – mivel az adatfüggőségek száma általában inkább konstans, mint a V függvényében változik – $O(J)$ műveletet jelent. Egy egyedi szelet kiszámítása egy gráfbejárás a DDG gráfon, aminek a komplexitását a kritériumból elérhető DDG élek száma határozza meg. Ez a legrosszabb esetben szintén $O(J \cdot V)$, de átlagosan – a fenti egyszerűsítéseket alkalmazva – csak $O(DEP)$. A táblázat első két DDG algoritmushoz tartozó sora ezek kombinációját tartalmazza: egy szelet kiszámítását DDG építés nélkül és azzal együtt. Mivel egyetlen gráfbejárással egyetlen szeletet határozunk meg, az összes (J darab) szelet kiszámításához legrosszabb esetben $O(J \cdot J \cdot V)$, átlagos esetben $O(J \cdot DEP)$ lépés szükséges.

A DDG alapú algoritmus tárigény komplexitása a gráf tárigénye, ami legrosszabb esetben $O(J \cdot V)$. Ez átlagos esetben – felhasználva, hogy egy pontból kiinduló élek száma átlagos esetben konstans – $O(J)$.

8.1.2. Igényvezérelt algoritmusok

A **hátramutató szeleteket számító algoritmus** számítási komplexitásánál figyelembe vesszük az *EHT* építését is. Az *EHT* táblát – ahogyan azt korábban leírtuk – arra használjuk, hogy a visszafelé haladó feldolgozásnál az *EH* elemeit hatékonyan tudjuk bejárni. A tábla $O(J)$ időben felépíthető. Az algoritmus számítási komplexitása a legrosszabb estben $O(J \cdot V \cdot \log(J))$, hiszen legfeljebb J iteráció lesz amelyekben egyenként legfeljebb V változót kell feldolgozni. Végül, az $O(\log(J))$ tag a különböző halmazműveletek miatt van jelen, nevezetesen az utolsó definiáló akció meghatározása *EHT* alapján és a *worklist* bővítése végezhető el ilyen időben. Átlagos esetben az iterációk száma (ami a *worklist*-ben megforduló elemek számával egyenlő) éppen az adott kritérium akciójából elérhető akciók átlagos *DEP* száma lesz. Továbbá az egy utasításhoz tartozó

használt változók átlagos száma inkább egy konstans, mintsem V függvénye. Így az átlagos számítási komplexitás – az *EHT* felépítésével együtt – $O(J + DEP \cdot \log(J))$.

Az algoritmus tárigénye tulajdonképpen a *worklist* plusz az *EHT* tábla mérete. Utóbbi miatt mind az átlagos mind a legrosszabb eset $O(J)$-nek adódik.

Az **előremutató szeleteket számító algoritmus** komplexitása ettől egy kicsiben eltér. Az algoritmusnak a legrosszabb esetben $O(J \cdot V)$ időre van szüksége, mert a legfeljebb J iteráció mindegyikében maximum V változót kell megjelölnie. Az egy utasításban használt változók számát a már megszokott módon konstansnak tekintve ez az érték átlagos esetben $O(J)$-re módosul.

A változók megjelölésének talán leghatékonyabb módja egy maximum $O(V)$ méretű vektor, amiből átlagos esetben $O(V^{DEF})$ méret is elegendő.

8.1.3. Praktikus algoritmusok

A praktikus globális algoritmusok esetén az eredmény-szeletek tárolását külön nem számítjuk, hiszen minden egyes szelet a kiszámításának pillanatában kiírható, azokra később már nincs szükség. Illetve amelyikre mégis, azt a megfelelő *DynDep* vagy *LiveAt* halmaz tartalmazni fogja.

A **hátramutató szeleteket számító praktikus algoritmus** számításigényét a végrehajtási nyom hossza (J) és a *DynDep* halmazokon végzett műveletek műveletigénye határozza meg.

Egy adott i^j lépésben a *DynDep* halmaz kiszámítása $O((|U_i| + D) \cdot \log(S))$ időt igényel, ahol $|U_i|$ az aktuális lépésben feldolgozott U_i halmaz számossága, D az összes felhasznált *DynDep* halmaz elemeinek száma, S pedig a feldolgozott akcióhoz tartozó szelet mérete. Nyilvánvaló, hogy $|U_i|$ felső korlátja V. A függőségi halmazok egyenként legfeljebb I elemet tartalmaznak és legfeljebb V ilyen halmaz van, tehát $D \leq I \cdot V$. Az algoritmus számítási komplexitása tehát a legrosszabb esetben $O(J \cdot I \cdot V \cdot \log(I))$. Ez átlagos esetben a következőképpen módosul. Mint azt már többször említettük $|U_i|$ általában konstansnak tekinthető, továbbá DS-sel jelölve a szeletek átlagos méretét, az algoritmus átlagos számítási komplexitása így $O(J \cdot DS \cdot \log(DS))$-nek adódik.

Az algoritmus tárigényét a függőségi halmazok száma és mérete határozza meg. Ez a legrosszabb esetben $O(V \cdot I)$, hiszen minden változóhoz tartozik egy-egy ilyen halmaz, és a halmaz legrosszabb esetben a program összes utasítását tartalmazhatja. De egy átlagos futás során csak V^{DEF} változót használ a program, és a szeletek átlagos mérete is csak DS, tehát az algoritmus átlagos tárigénye $O(V^{DEF} \cdot DS)$-nek adódik.

Az **előremutató szeleteket számító praktikus algoritmus** idő és tárigénye megegyezik a hátramutató szeleteket számító algoritmus komplexitásával, egy lényeges gya-

korlati különbséggel. A végrehajtási nyom fordított irányú feldolgozása sok esetben nehézségeket okozhat, plusz számítást és/vagy tárhelyet igényelve, sőt, az is előfordulhat, hogy az algoritmus alkalmazását teljesen ellehetetleníti. De ezzel most nem foglalkozunk.

Az algoritmus komplexitása tehát megegyezik a másik praktikus algoritmus komplexitásával, az ottani okfejtésben a *DynDep* halmazokat *LiveAt* halmazokra cserélve így a legrosszabb eseti és az átlagos számítási komlexitás $O(J \cdot I \cdot V \cdot \log(I))$-nek illetve $O(J \cdot DS \cdot \log(DS))$-nek adódik. A tárigény hasonló módon $O(V \cdot I)$ lesz a legrosszabb, és $O(V^{DEF} \cdot DS)$ az átlagos esetben.

8.1.4. Párhuzamos algoritmusok

A párhuzamos algoritmusok komlexitásai ugyanúgy megegyeznek egymással, mint a praktikus algoritmusok komplexitásai. Az algoritmusok komplexitását itt is a végrehajtási nyom J hossza, a *DynDep* illetve *LiveAt* halmazok, valamint a szeletek kibővítése határozza meg.

Az **előremutató szeleteket számító algoritmus** egy iterációjában a szeletek bővítése $O(L \cdot \log(S))$ időt igényel, ahol L a *LiveAt* halmaz mérete és S a szeletméret. A *LiveAt* halmazok kiszámolása $O(|U_i| \cdot L \cdot \log(L))$ időben történik. Az $L \leq J$, $S \leq I$ és $|U_i| \leq V$ felső korlátokkal a legrosszabb eset időigénye $O(J^2 \cdot (\log(I) + V \cdot \log(J)))$. Felhasználva, hogy a *LiveAt* halmaz mérete átlagosan DEP, $O(J \cdot DEP \cdot \log(DS \cdot DEP))$ jön ki átlagos számítási komplexitásként.

A tárigény lényegesen rosszabb, mint a praktikus algoritmusok esetében, hiszen az összes szeletet is tárolni kell az algoritmus teljes futása alatt. Az algoritmus tárigényét az EH hossza, az hátramutató szeletek mérete, a változók száma és a *LiveAt* halmazok mérete határozza meg. Ezekből a legrosszabb eset tárigénye $O(J \cdot (I+V))$-nek, az átlagos eseté $O(J \cdot DS + V^{DEF} \cdot DEP)$-nek adódik.

Az **előremutató szeleteket számító algoritmusban** egy iteráció $O(D \cdot \log(S) + D \cdot |U_i| \cdot \log(D))$ időt igényel, ahol D a *DynDep* halmaz mérete és S a szeletméret. A $D \leq J$, $S \leq I$ és $|U_i| \leq V$ felső korlátokat behelyettesítve a legrosszabb esetre itt is $O(J^2 \cdot (\log(I) + V \cdot \log(J)))$ adódik számítási komplexitásként, míg átlagos esetben – mivel a *DynDep* halmaz akciókat tartalmaz, és így az átlagos mérete DEP – ez szintén $O(J \cdot DEP \cdot \log(DS \cdot DEP))$ lesz.

Hasonlóan a másik párhuzamos algoritmushoz, a tárigényt most is az EH hossza, az előremutató szeletek mérete, a változók száma és a *DynDep* halmazok mérete határozza meg. Így a tárigény a legrosszabb esetben $O(J \cdot (I + V))$, átlagos esetben pedig $O(J \cdot DS + V^{DEF} \cdot DEP)$ lesz.

8.1.5. Komplexitások összehasonlítása

Mint azt a 6.3. fejezetben megmutattuk, az algoritmusaink ugyanazokat a szeleteket állítják elő, mint a DDG alapú algoritmus. A kérdés az, hogy idő és tárigény szempontjából hogyan viselkednek ezek az algoritmusok a DDG-hez képest. Az összehasonlítás során nem fogjuk figyelembe venni az algoritmusokhoz szükséges statikus számításokat. Ez a mi algoritmusaink esetén a $d : U$ reprezentáció, a DDG esetén pedig lehet az SDG – bár a cikkekben ez nincs pontosan letisztázva. Mindenesetre akármelyik reprezentációt használjuk is, mindkettőnek nagyjából ugyanannyi az előállítási költsége.

A következőkben az átlagos komplexitásokra koncentrálunk, hiszen az algoritmusok használhatóságát ezek határozzák meg. Továbbá a párhuzamos algoritmusokkal nem foglalkozunk, mert ezeknek a komplexitása mind idő- mind tárigény tekintetében nyilvánvalóan rosszabb, mint a DDG alapú algoritmusé. Ennek ellenére az előremutató párhuzamos algoritmus az előremutató praktikusnak egy valós alternatívája lehet a végrehajtási nyom feldolgozásának iránya miatt. (Ahogyan az a 7.1. táblázatban is látszik, egy egyszerűsített verzióját meg is valósítottuk Java nyelvre.)

Igényvezérelt algoritmusok

Először az igényvezérelt algoritmusaink idő- és tárigényét hasonlítjuk össze a DDG alapú algoritmuséval, egyetlen szelet kiszámítása esetén. Megállapításaink a következők. A DDG gráf felépítését nem számolva a gráfbejárás mindkét módszerünknél gyorsabb ($O(DEP)$ szemben a mi algoritmusaink $O(J)$ és $O(DEP \cdot \log(J))$ idejével). Ha azonban a DDG felépítését is az algoritmus részének tekintjük, akkor az előremutató szeleteket a mi $d : U$ alapú algoritmusunk implementáció függvényében akár gyorsabban is számolhatja, hiszen nincs szüksége a felépített gráf újbóli bejárására, hanem a gráf felépítése helyett egyből a bejárás eredményét számolja. Ez a különbség elegendő lehet arra, hogy megfelelő módon kiaknázzuk, de nem annyira lényeges, hogy akár egy kevésbé optimalizált implementáció esetén el ne tűnjön. A tárigényeket tekintve a hátramutató szeletelő algoritmusunkra ugyanez igaz. Vagyis, bár átlagos komplexitásnál mindkét helyen $O(J)$ van feltüntetve, a DDG-nél ez egy konstans szorzóval szerepel. Ez abból adódik, hogy ott az akcióknak megfelelő gráfpontokon túl egyszerre jelenik meg a statikus élek minden dinamikus megfelelője is, míg a $d : U$ alapú algoritmusnál ezek az „élek” csak a statikus $d : U$ reprezentáció által vannak jelen, és igény szerint számítódnak. Az előremutató szeletelésnél sokkal nyilvánvalóbb a $d : U$ alapú algoritmus hatékonysága, hiszen $V^{DEF} \leq J$, és az egyenlőség csak nagyon extrém esetekben teljesül. Ha figyelembe vesszük, hogy valós alkalmazásokban a V^{DEF}-nek ugyan potenciálisan J a felső korlátja – hiszen ilyenkor

a definiált változó fogalma a program futása során használt különböző memóriahelyeket jelöli –, de a gyakorlati mérések alapján az értéke mégis inkább V, mint J függvénye (mint azt a 8.2.1. fejezetben a 8.4. táblázat elemzésénél látni fogjuk), akkor a $d : U$ alapú algoritmus előnye még nyilvánvalóbb.

Praktikus algoritmusok

Most a praktikus algoritmusaink tárigényét fogjuk a DDG alapú algoritmus tárigényével összehasonlítani. Mint azt láttuk, az előremutató és hátramutató praktikus algoritmusok tárigénye ugyanaz (nem számítva a végrehajtási nyom visszafelé haladó feldolgozásából adódó pluszt). Fontos megjegyezni, hogy a komplexitások az eredményszeletek tárolása nélkül értendők mind a $d : U$, mind a DDG alapú algoritmusoknál (mint ahogyan a komplexitásokat eleve megadtuk). A DDG alapú algoritmus tárolási komplexitása $O(J)$, míg a $d : U$ alapú algoritmusé $O(V^{DEF} \cdot DS)$. Ez utóbbi adott programra nyilvánvalóan korlátos, míg J potenciálisan akármekkora lehet, tehát a $d : U$ alapú algoritmus használhatóbbnak tűnik. Ha itt is figyelembe vesszük, hogy valós alkalmazásokban a V^{DEF} értéke mégis inkább V, mint J függvénye, a $d : U$ alapú algoritmus a gyakorlatban is jobbnak bizonyul a DDG alapú párjánál.

A praktikus algoritmusaink időigénye nem kifejezetten jobb vagy rosszabb, mint a DDG algoritmusé; a kettő közötti reláció a program felépítésén múlik. Mindkét esetben van egy J szorzó, ami az összes szelet kiszámításának szól. A DDG esetében ennyiszer kell végrehajtani egy gráfbejárást átlagosan $O(DEP)$ időben, míg a $d : U$ alapú algoritmusnál a függőségi halmazok karbantartása $O(DS \cdot \log(DS))$ időt igényel. Mivel DS a program méretével arányos így korlátos, DEP viszont az EH méretével arányos tehát potenciálisan korlátlan, megfelelően nagy végrehajtási nyom esetén ismét csak a $d : U$ alapú algoritmus tűnik használhatóbbnak.

Konklúzió

Elméleti fejtegetéseink alapján tehát azt mondhatjuk, hogy a praktikus algoritmusaink megfelelő alternatívát kínálnak a DDG alapú algoritmus ellenében az összes dinamikus szelet kiszámítására mind számítási idő, mind memória felhasználás tekintetében, hiszen inkább függenek statikus, tehát korlátos értékektől, mint például a potenciálisan korlátlan EH hosszától. Továbbá a DDG felépítését is az algoritmus részének tekintve egyetlen szelet kiszámítására az igényvezérelt algoritmusainknak is kedvezőbbek a költségei, mint a DDG alapú algoritmusnak.

8.2. Mérési eredmények

Amint azt a 7. fejezetben leírtuk, egyes algoritmusokat implementáltunk különböző nyelvekre. Az implementációkkal többféle mérést is végeztünk. A Java implementáció esetében például elsődlegesen a szeletméretekre koncentráltunk [SGB07], míg a C szeletelőnél a használhatóság kérdése volt a fő szempont [BGG07].

8.2.1. Mérések C nyelvű implementációval

Ebben az alfejezetben a C nyelvű implementációval végzett méréseket közöljük. A mérések célja elsősorban az algoritmus gyakorlati használhatóságának igazolása volt.

Tesztprogramok, szeletméretek, változók száma

A két hátramutató szeleteket számító algoritmus (praktikus és igényvezérelt) C nyelvű implementációját öt kisebb programon mértük. Ezek a *bcdd*, *unzoo*, *bzip*, *bc* és *less*. A mérések során a tesztprogramok különböző tulajdonságait, és az algoritmusok komplexitásban megjelenő jellemzőit rögzítettük.

Program	utasítások száma	változók száma	használati halmaz elemszám	átlagos szeletméret
bcdd	78	179	5.4	18.27%
unzoo	932	1896	8.9	5.17%
bzip	2270	4184	8.1	4.35%
bc	3441	6898	12.6	3.37%
less	5373	10605	6.9	4.80%

8.3. táblázat. C tesztprogramok tulajdonságai

A 8.3. táblázatban a programok bizonyos jellemzőit foglaltuk össze. A táblázatban láthatjuk a program utasításainak számát (I). A *változók* oszlopban a statikus reprezentáció változóinak (a továbbiakban egyszerűen *statikus változók*ként hivatkozunk rájuk) száma látható, korábban ezt jelöltük V-vel. Ez – mint azt már említettük a korábbi fejezetekben – több különböző típusú változót takar. Vannak közöttük explicit memóriahivatkozások, mint skaláris változók, mutató dereferenciák, stb., valamint olyan virtuális változók, mint a predikátumváltozók vagy a függvények argumentumváltozói. A táblázatban látható még a $d : U$ reprezentáció használati halmazainak átlagos elemszáma, valamint a kiszámított szeletek átlagos mérete (a program utasításszámához viszonyítva). Mint látható, az átlagos szelet a teljes programnak csak egy igen kis részét

tartalmazza. A mérések során ellenőriztük, hogy vajon a program- illetve szeletméret és a végrehajtási nyom hossza között van-e bármilyen kapcsolat, és nem találtunk ilyet (ami várható eredmény volt).

Mint azt láttuk a 7. fejezetben, a C nyelvre történő implementációhoz szükség volt a *dinamikus változók* fogalmának bevezetésére, amiket tulajdonképpen a lefoglalt memóriaterületekkel azonosítunk. Az algoritmusunk így memóriaterületek közötti adatfüggőségekkel számol. Mivel az átlagos komplexitásban szerepe van, talán érdemes egy kicsit közelebbről megvizsgálni az ilyen dinamikus változók (korábban már V^{DEF}-fel jelölt) számát. A 8.4. táblázat első két oszlopa a statikus változók számát mutatja, pontosabban az explicit memóriahivatkozások számát (skaláris változók illetve dereferenciák). A 3-5. oszlopok a tesztfutások során keletkezett dinamikus változók számának minimumát, maximumát és átlagát mutatja. A statikus és dinamikus változók számának korrelációja viszonylag magas, 0.73. A statikus és dinamikus változók számának eltérése különféle, a korábbi fejezetekben már megemlített okokkal magyarázható. Az eredmények bíztatóak olyan szempontból, hogy a program változóinak memóriacímre cserélése nem okoz a változók számának megtöbbszöröződéséből adódó leküzdhetetlen problémát.

Program	statikus változók		dinamikus változók		
Program	skalárisok	dereferenciák	minimum	maximum	átlag
bcdd	55	4	34	34	34
unzoo	493	98	271	5664	1173
bzip	1040	194	82	1586	985
bc	1323	432	452	844	634
less	1860	402	1771	2667	2117

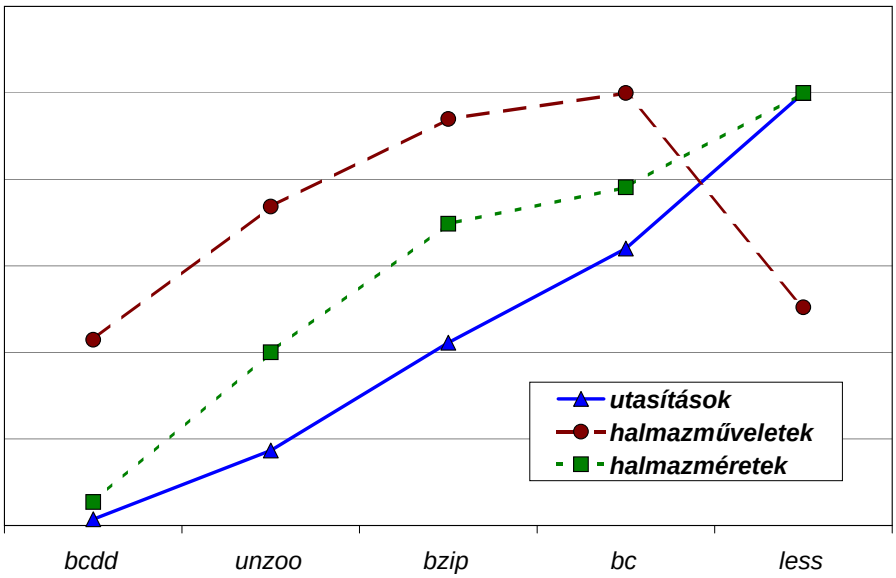
8.4. táblázat. Változók száma

Praktikus algoritmus mérései

A fő kérdésünk – amire az algoritmusaink komplexitásának meghatározásával kerestük a választ – az, hogy megéri-e őket használni a DDG alapú algoritmus helyett. Ezt praktikus algoritmusoknál leginkább a halmazműveletek száma határozza meg, ami pedig a halmazmérettől is függ. A kérdés tehát másképpen megfogalmazva: a halmazok mérete és a halmazműveletek száma vajon milyen viszonyban áll a program illetve a végrehajtási nyom méretével.

Először is megvizsgáltuk a praktikus hátramutató algoritmus összes lefutására a maximális halmazméreteket és a lépésenkénti halmazműveletek átlagos számát. A mért

értékek (normalizálva) a 8.1. ábrán láthatóak. Mint látható, a mért értékek a programmérettel együtt többé-kevésbé lineárisan változnak.



X *tengely: programok;* **Y** *tengely: relatív darabszám és halmazméret.*

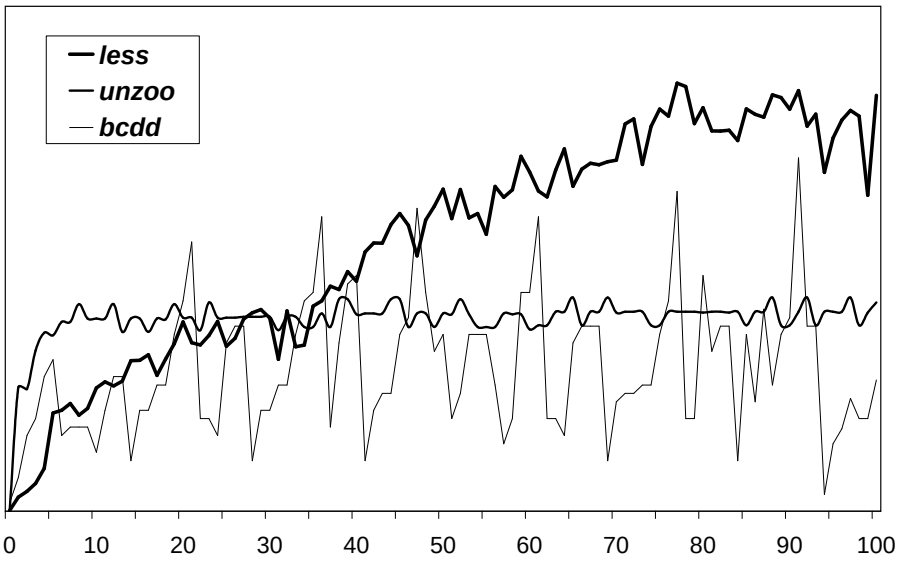
8.1. ábra. Halmazok mérete és halmazműveletek száma

Szintén mértük, hogy a halmazok maximális mérete és az összes halmazban lévő elemszám hogyan változik az algoritmus futása során a végrehajtási nyom feldolgozásával. A 8.3. ábrán látható háromszor kettő tipikus görbe látható. Mindegyik programhoz a maximális szeletméret és az összelemszám. A maximális méreteket normalizálva a 8.2. ábrára is külön kigyűjtöttük. Az x tengely az *EH* százalékos feldolgozottságát mutatja, az y tengelyen pedig (a programonkénti ábrákon) a halmazméret olvasható le. Mint látható, a három tipikus program különbözőképpen viselkedik, de a halmazok maximális mérete nem nő szignifikánsan a nyom feldolgozásának előrehaladtával.

Összességében tehát elmondhatjuk, hogy az algoritmus futási idejét leginkább meghatározó halmazműveletek száma inkább a program méretétől, sem mint a végrehajtásának hosszától függ.

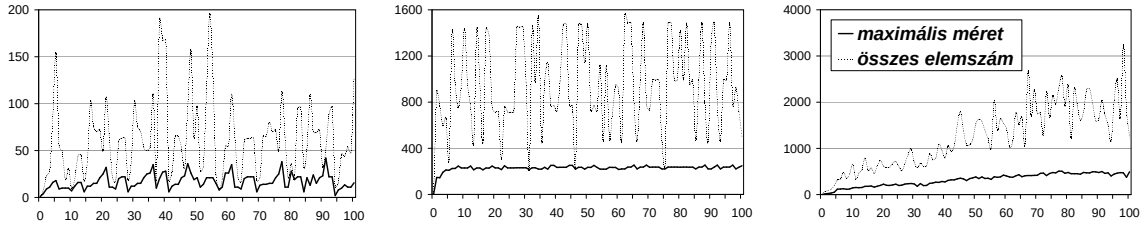
Igényvezérelt algoritmus

Az igényvezérelt algoritmus esetén az iterációs szám és a *worklist* mérete az, ami leginkább meghatározza a használhatóságot. Ezek a programfüggő értékek viszont nehezen becsülhetők. Ezeknél a méréseknél minden programhoz kiválasztottunk egy-egy tesztesetet, és annak az egyetlen esetnek az összes kritériumára futtattuk az igényvezérelt



X *tengely: végrehajtás %-a;* **Y** *tengely: relatív halmazméret.*

8.2. ábra. Maximális halmazméretek a feldolgozottság függvényében



(a) bcdd (b) unzoo (c) less

X *tengely: végrehajtás %-a;* **Y** *tengely: halmazméret.*

8.3. ábra. Maximális halmazméretek és elemszámok programonként

algoritmust.

A 8.5. táblázatban látható néhány, az iterációk számára vonatkozó adat. Az első két oszlop a program utolsó szeletelési kritériumát definiáló akció *EH*-beli sorszámát, és az ehhez tartozó iterációszámot, míg az utolsó oszlop az átlagos iterációszámot tartalmazza. Összehasonlítva ezt az értéket a végrehajtási nyom hosszával és a szeletméretekkel azt kaptuk, hogy ezeknek nincs igazán közük egymáshoz, vagyis például a hosszabb végrehajtási nyomból nem következik az algoritmus iterációs számának növekedése.

A 8.6. táblázatban a *worklist* méretére vonatkozó információk láthatók. Az első három oszlop a *worklist* maximális méretére vonatkozó adatokat tartalmaz, míg az utolsó

Program	utolsó kritérium lépések száma	utolsó kritérium iterációk száma	iterációk átlagos száma
bcdd	622	255	204
unzoo	11411	1564	482
bzip	8765	71	1185
bc	10669	5318	2454
less	38341	6944	3764

8.5. táblázat. Iterációs számok

oszlop az előbb említett utolsó kritériumhoz tartozó maximális halmazméretet mutatja. Látható, hogy ezek az értékek viszonylag alacsonyak a végrehajtás hosszához és az iterációk számához képest is. Mivel a *worklist* feldolgozatlan függőségeket tartalmaz, azt várnánk, hogy a mérete és a szeletméret között valamilyen kapcsolat lesz. Ezért korrelációt számoltunk a munkahalmaz mérete, a szelet mérete és a végrehajtási nyom mérete között, és azt kaptuk, hogy a munkahalmaz mérete nagymértékben a szeletek méretével korrelál.

Program	min.	max.	átlag	utolsó
bcdd	1	12	11	12
unzoo	1	220	13	18
bzip	1	49	18	10
bc	1	104	51	102
less	1	342	141	264

8.6. táblázat. A *worklist* mérete

Azt mondhatjuk tehát, hogy az igényvezérelt hátramutató szeleteket számító algoritmus valószínűleg jóval kevesebb iterációs lépésben fog lefutni, mint amennyi a végrehajtási nyom hossza, és hogy a *worklist* mérete nem jelent szűk keresztmetszetet az algoritmus futása során performancia szempontjából.

8.2.2. Mérések Java nyelvű implementációval

Ebben a fejezetben a Java implementációval végzett méréseinket foglaljuk össze. Ezek a mérések leginkább a különféleképpen előállított szeletek méretére vonatkoztak, nevezetesen a statikus, dinamikus és uniós szeletek viszonyára. A statikus szeletek előállításához az *Indus* [31] nevű Java statikus szeletelő eszközt használtuk.

Tesztprogramok

A mérésekhez összesen öt kis Java programot használtunk. A *RayTracer* a Java Grande Forum Benchmark Suite [33] része. A programot kicsit átalakítottuk, hogy fogadjon inputot is, és ne csak a két fix kép valamelyikét számolja ki. A *JSubtitles* [35] egy felirat átalakító, a *NanoXML* [46] egy kis XML elemző könyvtár, ami tartalmaz egy *DumpXML* nevű példaprogramot. A méréseinkhez ezt használtuk. A *java2html* [34] program Java forráskódot konvertál html, tex, rtf és más formátumokba szintaxis színezéssel, míg az ötödik a *dynjava* [22], egy dynamic java forráskód interpreter, ami a dynamic java language nyelven írt programok futtatására alkalmas.

A tesztprogramok néhány jellemzője a 8.7. táblázatban látható. Az osztályok, sorok (csak amelyekhez bájtkód generálódik) és a felhasznált tesztesetek számát tüntettük fel.

Tesztprogram	Osztályok	Sorok	Teszt esetek
RayTracer	12	340	20
JSubtitles	15	460	100
NanoXML	27	1 156	96
java2html	55	2 290	95
dynjava	302	17 447	25

8.7. táblázat. Java tesztprogramok jellemzői

Tesztesetek és lefedettség

A futtatásnál alkalmazott tesztesetek két részre oszthatók. Az első felükben preparált inputot használunk azért, hogy a programoknak kicsi, de jó elszeparált részeit futtassuk (ha ez lehetséges). A fent említett programokat kézzel megvizsgáltuk annak érdekében, hogy megfelelő inputokat gyárthassunk. A tesztesetek maradék része általános, valós inputtal futtatott teszteset. Néhány esetben a program mellé „csomagolt” teszteseteket, máskor egyéb forrásokból gyűjtött inputokat használtunk. Ezután az inputokat véletlenszerű sorrendbe rendeztük de úgy, hogy a preparált inputok minden esetben megelőzték a valós inputokat. A lista elemeinek sorrendje határozta meg az uniós szeletek számításánál az egyes dinamikus szeletek uniózásának sorrendjét.

A programokat azután lefuttattuk az összes inputjaikra a lista által meghatározott sorrendben. A 8.8. táblázat mutatja az egyes programok teszteseteinek futtatása során végrehajtott bájtkód utasítások legkisebb és legnagyobb számát.

A szeleteket a praktikus hátramutató szeleteket számító algoritmussal számoltuk ki minden értelmes kritériumra. Vagyis minden olyan utasításra, amelyek nem a `java.*`,

Program	Végrehajtott utasítások minimuma	maximuma
RayTracer	2 598 546	21 525 307 460
JSubtitles	516 213	55 459 126
NanoXML	910 806	94 754 237
java2html	1 541 531	20 370 505
dynjava	4 019 365	6 369 636

8.8. táblázat. Végrehajtott utasítások

`javax.*`, vagy `gnu.*` csomagok valamelyikéhez tartoztak. Az ilyen utasításokat a szeletekbe sem vettük bele.

A szeletelő eszközünk kimenete forráskód sorszámok halmaza, a szeletek méretét tehát kódsorban adjuk meg. A relatív szeletméreteket az abszolút szeletméret és az adott program összes sora számának hányadosaként kapjuk (és nem csak az aktuálisan betöltött osztályok sorainak számával osztunk). Végül, technikai okok miatt a lefedettséget bájtkód szintjén számoltuk, de az így kapott relatív értékek a relatív szeletméretekkel összehasonlíthatóak.

Statikus szeletek

A statikus szeletek meghatározásához az *Indus* [31] Java szeletelő és statikus analízis eszközöket használtuk. A rendszer sokféleképpen konfigurálható, az általunk választott (egyébként az egyik alapértelmezett) nem végrehajtható hátramutató szeleteket számító konfiguráció jellemzői: „non-termination sensitive", „intra- and interprocedural divergence analysis", „context sensitive deadlock criteria selection strategy", „symbol based ready- and interference dependencies".

Mivel az Indus egy *jimple* nevű köztes reprezentáción dolgozott, és az eredmények forráskódra történő visszavezetése hiányzott, a statikus szeletméret a jimple utasítások számában/arányában van meghatározva. A dinamikus szeleteléshez hasonlóan itt is kizártuk a könyvtári osztályokat a szeletszámításból.

A 8.9. táblázat első három oszlopa a statikus szeletek méreteiről ad statisztikai információkat. A *dynjava* programhoz memóriaproblémák miatt nem tudtuk statikus szeleteket számítani.

Dinamikus szeletek

Először minden teszteset minden kritériumához kiszámoltuk a hátramutató szeleteket, hogy megvizsgáljuk ezek méreteit. Mivel azonban a dinamikus kritérium eredeti definíciója szerint így minden egyes akcióhoz tartozott volna egy kritérium – ami túl

Program	átlag	szórás	max.
RayTracer	45%	37%	92%
JSubtitles	86%	24%	93%
NanoXML	25%	29%	59%
java2html	82%	21%	89%
dynjava	—	—	—

8.9. táblázat. Statikus szeletméretek, adatok

sok lett volna –, egy kicsit módosítottunk az elképzelésünkön, és statikus kritériumra számoltunk kvázi-dinamikus szeleteket, vagyis egy-egy utasítás többszöri előfordulását nem tekintettük külön kritériumnak. Ez gyakorlatilag azt jelentette, hogy az ugyanazon utasításhoz de különböző akciókhoz tartozó dinamikus szeleteket unióztuk. Előre és hátramutató szeleteket, és lefedettséget is számoltunk.

A 8.10. táblázat középső részén a mért hátramutató dinamikus szeletek átlagát, szórását és maximumát, valamint a kritériumok és a szeletek számát láthatjuk.

	szeletméret			krité-	
Program	átlag	szórás	max.	riumok	szeletek
RayTracer	9%	16%	58%	248	4 167
JSubtitles	6%	6%	36%	260	25 168
NanoXML	8%	5%	23%	652	58 428
java2html	3%	3%	16%	1 191	106 300
dynjava	2%	2%	7%	4 823	120 575

8.10. táblázat. Dinamikus szeletméretek, adatok

A *RayTracer* kivételével minden program nagyrészt különböző szeleteket mutatott a különböző bemenetek esetén. A kivétel program pedig egy benchmark része, és így teljesítménymérésre van kihegyezve, tehát számításigényes feladatot végez, de a vezérlési folyamot tekintve nem egy komplex program.

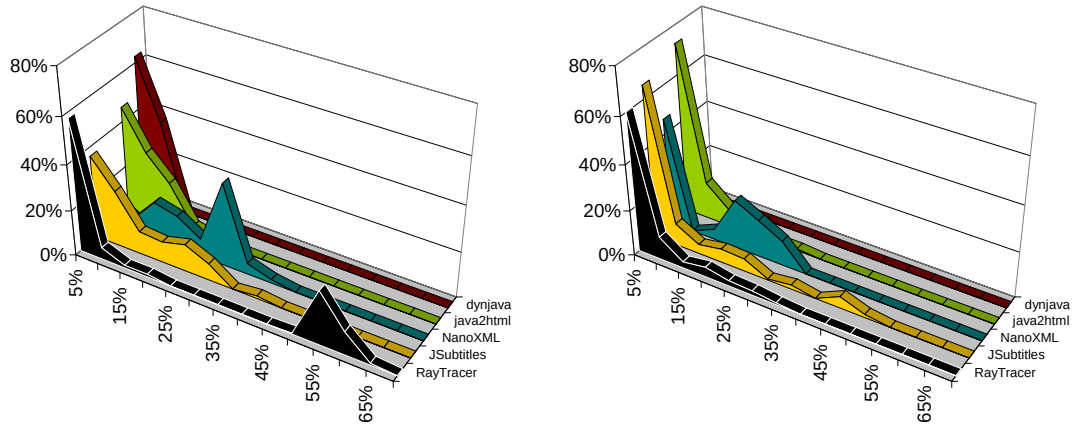
Uniós szeletek

Az uniós szeletek méréséhez hátra- és előremutató szeleteket, és *uniós lefedettséget* is számoltunk. Ez utóbbi azon bájtkód utasítások száma, amik legalább egyszer végrehajtódtak valamelyik teszteset futtatása során. A 8.11. táblázat utolsó három sorában láthatjuk az átlagos szeletméreteket és az említett lefedettséget. Az értékek szórása 3 és 9% között mozog, kivéve a *RayTracer*-t, aminél 25%. A maximális szeletméretek 9 és 60% között változtak. A *dynjava* programra technikai okok miatt nem sikerült előremutató szeleteket számolni.

Program	szeletméret		lefedettség
	előremutató	hátramutató	
RayTracer	20%	5%	87%
JSubtitles	11%	8%	72%
NanoXML	19%	10%	63%
java2html	6%	3%	61%
dynjava	4%	—	25%

8.11. táblázat. Uniós szeletméretek, adatok

A fenti számokból is látható, hogy az uniós szeletek jóval kisebbek, mint a statikus társaik. A 8.4. ábrán a hátra és előremutató uniós szeletek méretének hisztogramja látható. Binkley és Harman megállapították, hogy az előremutató statikus szeletek mérete kisebb, mint a hátramutató szeleteké. A méréseink alapján ez a megállapítás az uniós szeletekre is igaz.



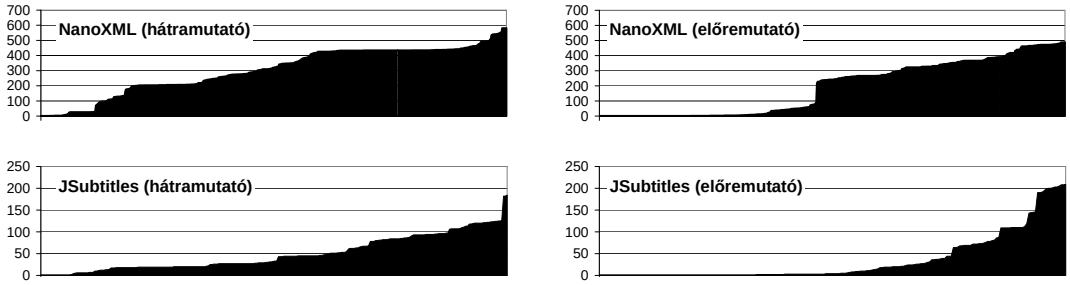
X *tengely: szelet relatív mérete;*
Y *tengely: adott méretű szeletek sránya;*
Z *tengely: programok.*

8.4. ábra. Uniós szeletméretek hisztogramjai

A 8.5. ábrán monoton szeletméret grafikonokat [14] láthatunk az előre és hátramutató szeletekre is. Mint látható, sokkal több a kis szelet található az előremutató szeletek között, de a maximális méretek nagyjából megegyeznek a hátramutató szeletek maximális értékeivel.

Uniós szeletek és a lefedettség növekedése

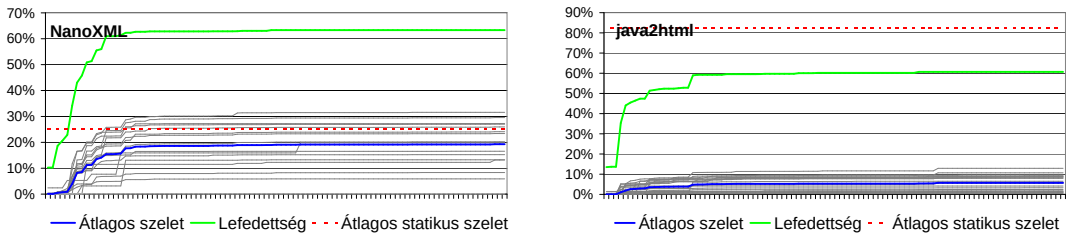
Az ugyanahhoz a kritériumhoz tartozó egyedi dinamikus szeleteket egymás után lépésről lépésre összegeztük, és megvizsgáltuk, hogyan nő az előállított uniós szelet mérete



X *tengely: szeletek, méret szerint rendezve;* **Y** *tengely: szeletméret.*

8.5. ábra. Monoton szeletméret grafikon uniós szeletekhez

tesztesetről tesztesetre a statikus szeletméret felé. Ugyanezt az összegzést a lefedettségre is megcsináltuk. Az eredményeket vizsgálva azt kaptuk, hogy mind a lefedettség, mind az uniós szeletméret több, mint 95%-a a lehetséges maximumnak. Két tesztprogram hátramutató szeleteinek reprezentatív eredményeit a 8.6. ábrán szemléltetjük.



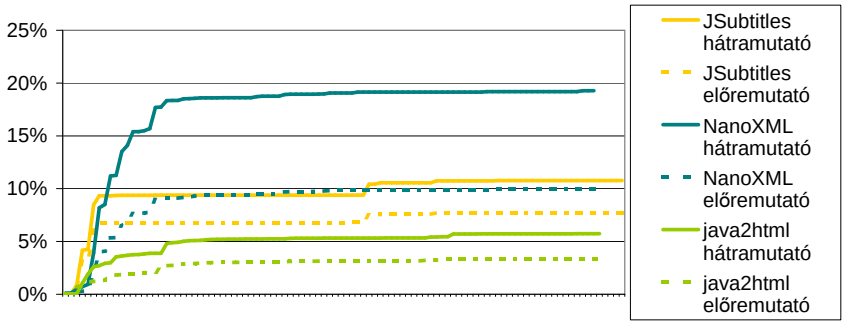
X *tengely: hozzáadott dinamikus szeletek száma;*
Y *tengely: szeletméret illetve lefedettség.*

8.6. ábra. Lefedettség és átlagos uniós szeletméret növekedés

Ami azonnal megfigyelhető, hogy az uniós szeletméretek görbéje a lefedettség görbéjét követi. A görbék között számított korreláció értéke 0.89 és 0.96 között mozgott. Ez jó hír, mert a lefedettség mértéke pontosan mérhető, az uniós szelet mérete a lehetséges legnagyobb méretéhez képest viszont csak becsülhető. Viszont a fentiek alapján a lefedettség felhasználható a becslésre. Így egy megfelelő teszteset-halmazzal az uniós szeletek kiválthatják a statikus szeleteket, és még egy kockázati értéket is tudunk mondani arra, hogy valamilyen – egyébként magvalósítható – függőséget nem vettünk figyelembe.

A görbék hasonlóságára vonatkozó megállapítás nem csak a hátramutató szeletekre igaz. A 8.7. ábrán látható három program előre és hátramutató szeleteinek növekedése. Az ábrán is megfigyelhető a fentebb már megállapított tény, hogy a hátramutató szeletek nagyobbak, mint az előremutató szeletek.

A *JSubtitles* és *java2html* programok szeleteinek vizsgálatánál azt tapasztaltuk, hogy



X *tengely: hozzáadott dinamikus szeletek száma;* **Y** *tengely: szeletméret.*

8.7. ábra. Hátra- és előremutató uniós szeletek méretnövekedése

a statikus és uniós szeletek közötti különbség meglehetősen nagy, a másik két program hasonló arányához képest. Ennek oka, hogy a programok felépítéséből adódóan a statikus szeletelés a maga konzervatív módján eleve nagy, pontatlan szeleteket számolt. Tehát van értelme a statikus szeleteket uniós szeletekkel közelíteni, hiszen az így kapott kisebb szelet az ember számára könnyebben feldolgozható.

8.3. Összefoglalás

Ebben a fejezetben kiértékeltük a $d : U$ alapú szeletelési algoritmusokat és implementációikat. Először az alap algoritmusok idő- és tárigény elemzésével arra jutottunk, hogy a praktikus algoritmusaink megfelelő alternatívát kínálnak a DDG alapú algoritmussal szemben az összes szelet kiszámításakor, illetve egy-egy szelet kiszámításához az igényvezérelt algoritmusaink is használhatóak. A C implementáció mérésének legfontosabb eredményei, hogy az algoritmusok futásidejét meghatározó tényezőknek (dinamikus változók száma, halmazműveletek száma) leginkább statikus összetevőik vannak, és az igényvezérelt hátramutató algoritmus lépésszáma jóval az *EH* mérete alatt van. A Java mérések kiértékelése során leginkább az uniós szeletekkel foglalkoztunk. Fontosabb eredmény, hogy az uniós szelet mérete jóval a statikus szelet mérete alatt van, illetve, hogy a növekedése (az újabb dinamikus szeletek uniózásával) erőteljesen korrelál a (szintén kumulált) lefedettség növekedéssel.

9. fejezet

Eredmények összegzése

A programok analízise egy nagyon szerteágazó kutatási terület, amelynek eredményeit számos formában fel lehet használni például program megértéshez, változás kiterjesztéshez vagy regressziós tesztelés során. A disszertációban ennek a nagy témakörnek két területével foglalkoztunk: a *hatásanalízissel* és a *programszeleteléssel*. A hatásanalízis során magas szintű programelemek között kell minél kevesebb időráfordítással kapcsolatokat felfedeznünk, míg a szeletelés alacsony szintű elemek között határoz meg pontos függőségeket.

A disszertációban először a hatásanalízissel kapcsolatos eredményeinket mutattuk be, amely a mások által definiált *Execute After* EA reláció kétféle bővítése/módosítása. Az egyik út a reláció statikussá tétele volt. Definiáltuk tehát a *Static Execute After* (SEA) relációt, és – mivel változás hatás analízisben a végrehajtási iránynak nincs igazán kitüntetett szerepe – rögtön a *Static Execute Before* (SEB) relációt is. A relációk hatékony kiszámításához megadtuk a program egy *Interprocedural Component Control Flow Graph* (ICCFG) reprezentációját, amely részletesebb a hívási gráfnál (ami önmagában nem volt elég), viszont kevésbé részletes mint egy interprocedurális vezérlési folyam gráf.

Az EA reláció kiterjesztésének másik módja a reláció korlátozása. Az eredeti reláció ugyanis a dinamikus hívási láncban adott módon egymásból elérhető eljárásokat kapcsol össze, de a kettő közötti út hosszára vonatkozó bármiféle hosszkorlát nélkül. Bevezettük tehát az eljárás-előfordulások közötti utak hosszát korlátozó $\mathrm{EA}^{(}(\mathrm{d})$, és párjaként az $\mathrm{EB}^{(}(\mathrm{d})$ (*Execute Before*), valamint uniójukként az $\mathrm{ER}^{(}(\mathrm{d})$ relációkat. A relációk alapján pedig meghatároztuk a függvénypárokon értelmezett *Dynamic Function Coupling* (DFC) metrikát. Megadtunk továbbá algoritmusokat, amelyek a metrikát illetve az általa meghatározott különféle hatáshalmazokat számolják. Az algoritmusokat implementáltuk, és konkrét méréseket is végeztünk az implementációval. A mérések alapján

a DFC által közelinek ítélt függvények között nagyobb valószínűséggel van kapcsolat, mint a távolinak ítéltek között. Továbbá meghatároztuk, hogy a DFC szerint legfeljebb 2 távolsággal csatolt elemeket a hatáshalmazba véve a hatáshalmaz mérete 65-95%kal csökken, ugyanakkor pontossága többszörösére nő.

A disszertáció második részében dinamikus programszeleteléssel foglalkoztunk. Először két már létező, a programábrázolásra $d : U$ reprezentációt használó algoritmus alapján további négy algoritmust készítettünk különböző szeletelési igényekhez. Megmutattuk továbbá, hogy az algoritmusok mindegyike ugyanazokat a szeleteket produkálja, mint a dinamikus szeletelésben egyfajta etalonként számon tartott DDG alapú szeletelés.

Ezután bemutattuk a szeletelő algoritmusok konkrét programnyelvre történő adaptálásának nehézségeit. Többek között a változók és pointerek, összetett adatstruktúrák, függvényhívás, kivételkezelés és többszálúság problémáját. Ezek közül az elsőre, azaz a változók és pointerek kezelésére C nyelvű szeletelés esetén megoldást is adtunk. A megoldás tulajdonképpen a változónevek memóriacímekre fordítása, amihez az instrumentált kódból ki kell nyernünk például a változóneveket, pointer értékeket, blokk belépési/kilépési eseményeket. Ezután vermek és a végrehajtási nyomból jövő pointer-adatok segítségével az algoritmus futása közben mindig pontosan azonosítható a változóhoz rendelt memóriacím. Pointerek segítségével a C egyéb adatszerkezetei, például a tömbök és a struktúrák is kezelhetőek.

Végül bemutattuk a szeletelő algoritmusok idő- és tárigényére vonatkozó számításainkat, valamint a C és Java nyelvű implementációval készült méréseinket. Ezek alapján az algoritmusaink praktikus alternatívát nyújthatnak a DDG alapú szeleteléssel szemben. Bár idő és tárigényük nem minden esetben jobb egyértelműen a DDG alapú algoritmusénál, a bennük szereplő tagok a gyakorlatban inkább a statikus (így korlátos) értékekkel korreláltak, mint a dinamikus (és így potenciálisan korlátlan) értékekkel. Vizsgáltuk továbbá az uniós szeletek kérdését, és a mérési eredmények azt támasztották alá, hogy az uniós szeletek jóval hatékonyabbak (kisebbek) statikus társaiknál. Továbbá teljességük a lefedettség által becsülhető, tehát tervezhető és pontosabb alternatívát nyújtanak a statikus szeleteléssel szemben.

Függelékek

A. Függelék

Magyar nyelvű összefoglaló

Bevezetés

A programok analízise (programelemek közötti kapcsolatok felderítése) rengeteg dologra felhasználható a szoftver életciklusa során, például teszteset-szelekcióra, nyomkövetésre, hibakeresésre, program megértésre, visszatervezésre vagy változás kiterjesztéshez. A kutatási terület nagy és szerteágazó, de bizonyos szempontok szerint számos területét meg lehet különböztetni egymástól. Egy ilyen szempont az, hogy az analízis csak statikusan (azaz a program végrehajtása nélkül) elérhető információkra támaszkodik, vagy felhasznál a program futtatásából származó dinamikus információkat is. Előbbit statikus, utóbbit dinamikus elemzésnek nevezzük. Egy másik kategorizálási szempont az, hogy magas szintű programelemek (például eljárások, metódusok, osztályok) között keressük a kapcsolatokat, vagy alacsony szintű elemek (tipikusan forráskódú vagy gépi kódú utasítások) között. A *hatásanalízis* az első kategóriába tartozik, míg a *programszeletelés* a második kategóriába. A disszertációban a programanalízisnek ezzel a két területtel foglalkoztunk.

Hatásanalízis

(A szerző kapcsolódó eredményei a [BGF07] és [BGJ07] publikációkban találhatók.)

A hatásanalízis [7] magas szintű programelemek (eljárások, függvények, metódusok, osztályok) közötti függőségek felderítésével foglalkozik. Erre a legpontosabb módszer nyilvánvalóan egy utasításszintű szeletelés eredményeinek eljárás- vagy osztályszintre vetítése lenne, de ez a szeletelés magas számítási igénye miatt nem éri meg. A hatásanalízisben tehát a valódi függőségek közelítésére különféle heurisztikákat használnak, amelyek nélkülözik a részletes analíziseket, és ezáltal bizonyos függőségeket nem tudnak

felfedezni. Ezeket nevezzük *rejtett függőségeknek* [64].

Részben a rejtett függőségek felfedezésének céljából definiáltuk a dinamikus *Execute After* (EA) [5] reláció statikus párjait, a *Static Execute After* (SEA) és *Static Execute Before* (SEB) relációkat. A relációk eljárás szinten vannak definiálva, elvük pedig, hogy az egymás után futtatott eljárások potenciálisan hatással lehetnek egymás működésére. A relációk biztonságosak, mert a fenti konzervatív megközelítés egyetlen potenciális (akár rejtett) függőséget sem hagy ki. A SEA/SEB számításához szükségünk van egy, az egyszerű hívási gráfnál részletesebb, de az *Interprocedural Control Flow Graph*-nál (ICFG) kevésbé részletes programábrázolásra. Definiáltuk tehát az *Interprocedural Component Control Flow Graph* (ICCFG) gráfot, ami egy-egy metódust egy belépési pontból, hívási helyekből és közöttük futó vezérlési folyam élekből álló körmentes gráfként ábrázol, a metódusok között pedig hívási élek futnak a hívási helyektől a megfelelő metódus belépési pontjáig.

Az EA relációval kapcsolatban problémának értékeltük azt a tulajdonságát, hogy nincs korlátozva két relációban lévő függvény távolsága. Ezért definiáltuk a *Dynamic Function Coupling* (DFC) metrikát, amelynek alapja az EA reláció, de bevezeti a távolság fogalmát. Nevezetesen, két EA relációban álló függvény DFC távolsága a dinamikus hívási fában köztük lévő út hossza. Megadtuk továbbá a DFC metrikát kiszámító, illetve (a rész-relációi alapján) hatáshalmazokat számító algoritmusokat is. Ezeket az algoritmusokat implementáltuk, majd felhasználtuk a metrika gyakorlati kiértékelésénél. Ezek alapján a DFC hasznos, hiszen az által közelinek ítélt függvények között nagyobb valószínűséggel van kapcsolat, mint a távolinak ítéltek között. A mérések azt is mutatták, hogy a DFC által alacsony távolságkorlát mellett számolt hatáshalmazok pontossága többszöröse a eredeti EA reláció alapján számolt halmazok pontosságának, bár teljességük alacsony. Ez viszont kisebb halmazméretet jelent, ami egy ember számára pozitív eredmény. Ezek alapján a DFC hatékonyan alkalmazható a hatásanalízis területén.

Programszeletelés

(A szerző kapcsolódó eredményei a [BGS01], [FG02], [BGG06] és [SGB07] publikációkban találhatók.)

A programszeletelés [59] alacsony szintű programelemek közötti pontos összefüggéseket szolgáltat. Jóval pontosabb és részletesebb eredményt ad, mint a hatásanalízis, viszont sokkal nagyobb a számításigénye is. A disszertációban csak dinamikus szeleteléssel [3] foglalkozunk. Azon belül is az úgynevezett *definíció-használat* ($d : U$) programábrázolást használó algoritmusokkal. Szerzőtársaim két korábbi $d : U$ alapú szele-

telő algoritmusát [28, 9] három tulajdonság alapján (szeletek iránya, feldolgozás iránya, egy bejárással kiszámolt szeletek száma) kategorizáltuk, majd megadtunk további négy, különböző szeletelési igényekhez tartozó algoritmust. Ezekről megmutattuk, hogy ugyanazokat a szeleteket produkálják, mint a széles körben elfogadott *Dynamic Dependence Graph* (DDG) alapú algoritmus [3]. Az idő- és tárigények elemzése, valamint az ehhez kapcsolódó, C nyelvű implementációval végzett mérések azt mutatták, hogy az algoritmusaink nagy rendszereken is használható alternatívát kínálnak a DDG alapú algoritmussal szemben.

Az elméleti algoritmusok implementációja során rengeteg olyan technikai probléma merül fel, amely megakadályozhatja azok gyakorlati megvalósítását. Egy ilyen probléma például a mutatók és dinamikus változók lekezelése C nyelven. Az azonos szintaktikus hivatkozással rendelkező elemek ugyanis a futás során különböző memóriacellákra mutathatnak, és a különböző hivatkozások is mutathatnak ugyanarra a memóriaterületre. Ezért bevezettük a *dinamikus* $d : U$ fogalmát, ami nem (csak) szimbolikus nevekkel, hanem ahol lehet valódi memóriacímekkel dolgozik. A valódi memóriacímeket és a karbantartásukhoz szükséges információkat ugyanúgy mint a végrehajtási nyomot, a szeletelendő program instrumentálásával és végrehajtásával kaptuk meg. Ezen információk, és egy veremszerű adatszerkezet segítségével megoldottuk a statikus és dinamikus $d : U$ ábrázolás összekötését. Ezáltal megoldottuk a C nyelv változóinak, mutatóinak, struktúráinak és tömbjeinek kezelését is. Megmutattuk még az esetleg rekurzív függvényhívások esetén az adatkapcsolatok kezelését is. A C nyelvű implementáció segítségével valódi méréseken keresztül megmutattuk, hogy algoritmusaink valós környezetben is használhatóak. Ehhez mértük az algoritmusok jellemzőit (változók száma, halmazméretek, lépések száma). Azt kaptuk, hogy a futásidőt meghatározó mennyiségek inkább a programra jellemző konstansok, mint a végrehajtás hosszától függő értékek.

Végül a Java nyelvű implementációt arra használtuk, hogy kiértékeljük az uniós szeleteket, mint a statikus szeletek egy gyakorlati közelítését. A mérések azt mutatták, hogy az uniós szeletméret és a lefedettség között magas (0.9 körüli) a korreláció. Mivel ez utóbbi pontosan mérhető, uniós szeletekkel közelítve a statikus szeletet (nagy valószínűséggel) meg tudjuk mondani, hogy milyen messze vagyunk az elméleti statikus (úgynevezett realizálható) szelettől.

Konklúziók

A disszertációban tehát megadtunk két, a hatásanalízisben használható módszert. Az első használhatósága az általa használt tömör programábrázoláson alapszik. A második

használhatóságát gyakorlati mérésekkel igazoltuk.

A programszeletelésben kategorizáltuk a $d : U$ alapú két korábbi és négy újonnan definiált algoritmust. Elméleti módszerekkel és gyakorlati mérésekkel megmutattuk, hogy ezek valódi alternatívái a DDG alapú módszernek, és nagy rendszereken is használhatóak. Bemutattuk az algoritmusok C implementációja során a mutatók használatából fakadó problémát és adtunk rá egy megoldást. A Java implementációt uniós szeletek gyakorlati kiértékelésére használtuk fel.

B. Függelék

Summary in English

Introduction

Program analysis (detecting relationships between program elements) can be used for many purposes during the software's life-cycle, for example for test case selection, debugging, error location, program understanding, reverse engineering, or change propagation. The research area is large and diversified, but many of its fields can be distinguished from each other by certain aspects. One of these aspects is whether the analysis can only rely on statically available information (thus without executing the program), or it can use dynamic information acquired during the execution of the program too. The former is called static, the latter dynamic analysis. Another point of view of the classification is whether the relations are defined between high level program elements (e. g. procedures, methods, classes) or between low level elements (source code or assembly instructions, typically). *Impact analysis* belongs to the first category, while *program slicing* to the second one. In this dissertation we have discussed these two fields of program analysis.

Impact analysis

(Related results of the author can be found in the publications [BGF07] and [BGJ07].)

Impact analysis [7] deals with the detection of the dependencies between high level program elements (procedures, functions, methods, classes). Obviously, the most precise method to do this would be to lift the results of an instruction level program slicing to procedure or class level, but it is not worth it because of the high computational requirements of slicing. Thus, different heuristics are used in impact analysis to approximate the real dependencies, which cannot detect certain dependencies due to the lack of detailed analysis. These dependencies are called *hidden dependencies* [64].

Partially motivated by the detection of hidden dependencies, we defined the *Static Execute After* (SEA) and *Static Execute Before* (SEB) relations, the static counterparts of the dynamic *Execute After* (EA) [5] relation. The relations are defined on procedure level, and their basic idea is that procedures executed one after the other, may potentially affect each others' behaviour. The relations are safe, because the above conservative approach does not miss any potential (even hidden) dependencies. To compute the SEA/SEB relations, we need a program representation which is more detailed than a simple call graph, but less detailed than an *Interprocedural Control Flow Graph* (ICFG). Thus, we defined the *Interprocedural Component Control Flow Graph* (ICCFG), that represents each method as a directed acyclic graph consisting of a method entry node, some call site nodes and control flow edges between them, and there are interprocedural call edges from the call sites to the entry node of the corresponding method.

Regarding the EA relation, the fact that the distance between two functions in relation is not limited was treated as a problem. So, based on the EA relation we defined the *Dynamic Function Coupling* (DFC) metric, that utilizes distances. Namely, the DFC distance of two functions that are in EA relation is the length of the path between them in the dynamic call tree. Furthermore, we created algorithms that compute the DFC values and (based on the sub-relations of DFC) impact sets. We implemented these algorithms, and used the implementations to empirically evaluate the metric. Based on the results the DFC is proven to be useful, as the closer the two functions are to each other based on their DFC value, the more probable a true dependency is between them. Measurements also showed that the precision of the impact sets computed using DFC with a low distance threshold is many times greater than that of the sets computed by employing the original EA relation, although their completeness is low. On the other hand it results in smaller set sizes, which is a positive result for humans. Based on the above, the DFC can be effectively used for impact analysis.

Program slicing

(Related results of the author can be found in the publications [BGS01], [FG02], [BGG06] and [SGB07].)

Program slicing [59] provides exact relations between low level program elements. It produces much more precise and detailed results than impact analysis, however it requires much more computation. In this dissertation we discuss only dynamic slicing [3]. Within dynamic slicing, we deal with algorithms using the so-called *definition-use* ($d : U$) program representation. We classified two former slicing algorithms [28, 9] of my

co-authors based on their three properties (direction of slices, direction of processing, number of slices computed by one traversal), then we provided four new algorithms for different slicing scenarios. We showed that these algorithms compute exactly the same slices as the widely accepted method based on *Dynamic Dependence Graph* (DDG) [3]. Based on the results of analyzing time and space requirements, and on the empirical results of the related C implementation we can say that our algorithms are usable on large systems and are shown to be useful alternatives to the DDG-based algorithm.

During the implementation of the theoretical algorithms many technical problems arise, that could hinder their practical adaptation. One such problem is, for instance, the handling of pointers and dynamic variables in C language. Namely, the same syntactic elements can refer to different memory locations during the execution, and different elements can point to the same memory locations. Thus, we introduced the concept of *dynamic* $d : U$, that works not only with symbolic names, but with real memory addresses whenever it is possible. The real memory addresses and other information required to maintain them, as well as the execution history were acquired by instrumenting and executing the program to be sliced. Using this information and a stack-like data structure we solved the interconnection of static and dynamic $d : U$ representation. Thereby, we managed to handled the variables, pointers, structures and arrays of the C language too. The handling of dataflow through possibly recursive function calls was also shown. Through real measurements made with the implementation that slices C programs, we also showed that our algorithms are applicable in live environment. To this end we measured some traits of the algorithms (number of variables, set sizes, number of iterations). The results showed that quantities that determine the time requirements of the algorithms are rather constants specific to the program than values depending on the length of the program execution.

Finally, we used the Java implementation to evaluate union slices, the practical approximation of static slices. The results showed a high correlation (about 0.9) between union slice sizes and code coverage. As the latter can be measured exactly, we can tell (with a high probability) how far the union slices are from the corresponding theoretical static (so-called realizable) slices, when union slices are used to approximate static slices.

Conclusions

In the dissertation we gave two impact analysis methods. The usability of the first one is based on the concise program representation it uses. The usability of the second method was verified through empirical measurements.

In program slicing we classified two former and four new slicing methods that use $d : U$ program representation. Through theoretical methods and practical measurements we showed that these are real alternatives to the DDG-based method, and can be used on large systems too. We presented the problems deriving from the use of pointers during the implementation of the algorithms for C, and we provided a solution to it. The Java implementation was used to empirically evaluate the concept of union slices.

Irodalomjegyzék

[1] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1992.

[2] Hiralal Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 302–312, 1994.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[4] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, Montreal, Canada, September 1993.

[5] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 432–441, May 2005.

[6] Erik Arisholm, Lionel C. Briand, and Audun Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[7] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[8] J. Beck. Program and interface slicing for reverse engineering. In *Proceeding of the Fifteenth International Conference on Software Engineering*, 1993. Also in *Proceedings of the Working Conference on Reverse Engineering*.

[9] Árpád Beszédes. *Forráskód Analízis és Szeletelés a Programmegértés támogatásához*. PhD thesis, Szegedi Tudományegyetem, Matematika- és Számítástudományok Doktori Iskola, Szeged, November 2004.

[10] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–21. IEEE Computer Society, October 2002.

[11] Árpád Beszédes, Tibor Gyimóthy, Gábor Lóki, Gergely Diós, and Ferenc Kovács. Using backward dynamic program slicing to isolate influencing statements in GDB. In *Proceedings of the 2007 GCC Developers' Summit*, pages 21–30, July 2007.

[12] David Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[13] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996. Marvin Zelkowitz, Editor, Academic Press San Diego, CA.

[14] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 177–186, September 2005.

[15] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[16] Ben Breech, Anthony Danalis, Stacey A. Shindo, and Lori L. Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, pages 453–457, September 2004.

[17] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[18] Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 475–482, September 1999.

[19] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Václav Rajlich. JRipples: A tool for program comprehension during incremental change. In *Proceedings*

of the 13th International Workshop on Program Comprehension (IWPC'05), pages 149–152, May 2005.

[20] Andrea De Lucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'01)*, pages 142–149, November 2001.

[21] Andrea De Lucia, Mark Harman, Robert Hierons, and Jens Krinke. Unions of slices are not slices. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 363–367. IEEE Computer Society, March 2003.

[22] Homepage of dynjava. `http://koala.ilog.fr/djava/index.html`.

[23] Rudolf Ferenc, Árpád Beszédes, Ferenc Magyar, and Tibor Gyimóthy. A short introduction to columbus/can. Technical report, University of Szeged, 2001.

[24] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimóthy. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, 1992.

[25] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[26] Homepage of GNU GDB. `http://www.gnu.org/software/gdb/`.

[27] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, 1992.

[28] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*, number 1687 in Lecture Notes in Computer Science, pages 303–321. Springer-Verlag, September 1999.

[29] Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, March 1995.

[30] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[31] Indus project: Java program slicer and static analyses tools. `http://indus.projects.cis.ksu.edu/`.

[32] Homepage of JamVM. `http://jamvm.sourceforge.net`.

[33] Homepage of Java Grande Forum Benchmark Suite. `http://www.epcc.ed.ac.uk/javagrande/index_1.html`.

[34] Homepage of java2html. `http://www.java2html.de/`.

[35] Homepage of JSubtitles. `http://sourceforge.net/projects/jsubtitles`.

[36] Mariam Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, December 1995.

[37] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the 4th International Conference on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[38] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 118–127. IEEE Computer Society, September 2003.

[39] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[40] Bogdan Korel and Janusz W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[41] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103. ACM Press, January 1991.

[42] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 308–318, May 2003.

[43] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[44] Áine Mitchell and James F. Power. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. *Science of Computer Programming*, 59(1-2):4–25, 2006.

[45] G. B. Mund and Rajib Mall. An efficient interprocedural dynamic slicing method. *The Journal of Systems and Software*, 79(6):791–806, 2006.

[46] Homepage of NanoXML. `http://nanoxml.cyberelf.be/`.

[47] Jim-Karl Ng. Context-sensitive control flow graph. Master's thesis, Iowa State University, Ames, Iowa, USA, 2004.

[48] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering held jointly with 9th European Software Engineering Conference (ESEC/FSE'03)*, pages 128–137, September 2003.

[49] Václav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, 2004.

[50] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 432–448, October 2004.

[51] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of ISSTA'94*, pages 169–183, Seattle, Washington, August 1994.

[52] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[53] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[54] B. G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[55] Attila Szegedi and Tibor Gyimóthy. Dynamic slicing of Java bytecode programs. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 35–44. IEEE Computer Society, September 2005.

[56] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[57] Paolo Tonella, Giulio Antoniol, Roberto Fiutem, and Ettore Merlo. Variable precision reaching definitions analysis for software maintenance. In *Proceedings of the First Euromicro Conference on Software Maintenance and Reengineering*, pages 60–67, March 1997.

[58] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, March 1995.

[59] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[60] Lee White and Khalil Abdullah. A firewall approach for the regression testing of object-oriented software. In *10th International Software Quality Week (QW'97)*, page 27, May 1997.

[61] Lee White, Khaled Jaber, and Brian Robinson. Utilization of extended firewall for object-oriented regression testing. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 695–698, September 2005.

[62] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[63] Hong Yul Yang, Ewan Tempero, and Rebecca Berrigan. Detecting indirect coupling. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 212–221, 2005.

[64] Zhifeng Yu and Václav Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*, pages 293–299, May 2001.

[65] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, Washington, D. C., June 2004.

[66] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, Edinburgh, United Kingdom, May 2004.

[67] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27(4):631–661, July 2005.

Saját irodalom

[BGF07] Árpád Beszédes, Tamás Gergely, Szabolcs Faragó, Tibor Gyimóthy, and Ferenc Fischer. The dynamic function coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 103–112. IEEE Computer Society, March 21–23, 2007.

[BGG06] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 21–30. IEEE Computer Society, September 27–29, 2006.

[BGG07] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Investigation of graph-less dynamic program slicing algorithms. Technical report, University of Szeged, 2007.

[BGJ07] Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 295–304. IEEE Computer Society, October 2–5, 2007.

[BGS01] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 14–16, 2001. Best paper of the conference.

[FG02] Csaba Faragó and Tamás Gergely. Handling pointers and unstructured statements in the forward computed dynamic slice algorithm. *Acta Cybernetica*, 15:489–508, 2002.

[SGB07] Attila Szegedi, Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, and Gabriella Tóth. Verifying the concept of union slices on Java programs. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 233–242. IEEE Computer Society, March 21–23, 2007.