Forráskód analízis és szeletelés a programmegértés támogatásához

Beszédes Árpád
egyetemi tanársegéd
SZTE Szoftverfejlesztés Tanszék

Szeged, 2004. november

Témavezető:
Dr. Gyimóthy Tibor

Értekezés doktori fokozat
megszerzéséhez

Szegedi Tudományegyetem
Matematika- és Számítástudományok Doktori Iskola
Informatikai Doktori Program

„A felfedezés lényege: látni azt, amit már mindenki látott, de olyat gondolni, amit senki más nem gondolt róla.”

— Szent-Györgyi Albert

Előszó

A természettudományokban minden probléma megoldásához két dologra van szükség: megfelelő módszerre, és az azt támogató eszközökre. Így van ez a szoftverfejlesztés területén is, amikor azzal a problémával állunk szemben, hogy a szoftverrendszert megvalósító forráskódból további hasznosítható információt kell kigyűjteni. A forráskód célja a szoftver megvalósítása, és ha a megvalósított működést értelmeznünk kell (például mert különbözik az elvárttól, vagy tovább kell fejleszteni), a *forráskódot* kell értelmeznünk. Ezt pedig úgy végezzük, hogy a forrást megfelelő módon analizáljuk, kigyűjtjük belőle a szükséges információkat, valamint a megértést elősegítő további modelleket származtatunk. Az alkalmazási céltól függően ezen modellek és használatuk képezik a megértés módszerét, amelyhez viszont ki kell dolgozni a szükséges analizálási, modell létrehozási és származtatási algoritmusokat, eszközöket. Jelen értekezést is ezzel a gondolattal indítjuk, majd eredményként megadunk egy programmegértést támogató *forráskód analizátort* C/C++ nyelvekre és hatékony megoldásokat egy speciális analizálási technikához, a *dinamikus programszeleteléshez*.

Felépítés. A dolgozat felépítése is követi e két eredményt. Az első részben áttekintjük a forráskód analízisének problematikáját különös tekintettel a megértéshez szükséges elvárásokra, majd megadjuk az általunk kidolgozott C/C++ forráskód analizátor megvalósításának technológiáját és alkalmazásainak körét. A második rész foglalkozik a programszeletelés témájával. A kutatási terület rövid áttekintése után megadjuk az általunk kidolgozott újszerű megoldásokat a dinamikus szeletek kiszámítására valós C programokhoz, amit kiegészítünk részletes mérési eredményekkel is. Végül a dinamikus szeletelés alkalmazásaival és a konklúziókkal zárjuk az értekezést. Látni fogjuk, hogy mindkét témában újszerű megoldásokat alkalmaztunk, melyek a korábbiakhoz képest jobban támogatják a programmegértési folyamatokat.

Köszönetnyilvánítás. Először is, témavezetőmnek, Dr. Gyimóthy Tibornak tartozom köszönettel, amiért rávezetett a szoftverfejlesztés tárgykör eme izgalmas területeire, és hogy egyáltalán elindított a számítástudomány útján. Hálás vagyok azért, hogy a nálunk végzett kutatás mindig kézzelfogható, gyakorlati problémák megoldását jelenti, egy jó munkaközösségben. Köszönöm a segítséget munkatársaimnak, különösképpen Ferenc Rudolfnak, valamint hallgatóinknak a módszerek implementációjában és tesztelésében végzett munkájukért.

Végül, de nem utolsósorban szívből jövő köszönetemet fejezem ki a feleségemnek, kisfiamnak és az egész családomnak, amiért oly sokat nélkülöztek éjjel-nappal amíg e munkán dolgoztam.

Beszédes Árpád, 2004. november

Ábrák jegyzéke

3.1. A C/C++ front end felépítése 16
3.2. A C/C++ elemző architektúrája 17
3.3. Az elemző szintaktikus akció-interfésze 20

6.1. Példaprogram és annak dinamikus szelete az $(\langle \mathtt{a} = 0, \mathtt{n} = 2\rangle, 12^{15}, \{\mathtt{s}\})$ kritériumhoz 40
6.2. A példaprogram D/U reprezentációja 43
6.3. Globális algoritmus 44
6.4. A globális algoritmus példa futása 45
6.5. Igényvezérelt algoritmus 48
6.6. A példaprogram EHT táblája 50
6.7. Az igényvezérelt algoritmus példa futása 50
6.8. Függőségi DAG a példaprogramhoz 51

7.1. Az offline megvalósítás vázlata 59
7.2. C példaprogram és annak D/U programreprezentációja 62
7.3. A végrehajtási nyom formális szerkezete 70
7.4. Az eredeti és az instrumentált forráskód, valamint néhány instrumentáló függvény 72
7.5. A globális algoritmus C-re (főprogram) 79
7.6. A globális algoritmus C-re (függvény-feldolgozás és címfeloldás) 80
7.7. A globális algoritmus C-re (akció-feldolgozás) 81
7.8. Az igényvezérelt algoritmus C-re (főprogram) 82
7.9. Az igényvezérelt algoritmus C-re (akció-feldolgozás) 83

8.1. Szeletek méretei 89
8.2. A globális algoritmus komplexitási tényezői a programméret mellett 93
8.3. A halmazméretek változása a végrehajtás során 95
8.4. A szeletek méretével való összehasonlítás 99

9.1. Példaprogram a dinamikus és releváns szeleteivel a $(\langle \mathtt{a} = 0, \mathtt{n} = 2\rangle, 14^{19}, \{\mathtt{s}\})$ kritériumhoz 103
9.2. A realizálható szelet közelítése 106
9.3. A szelet-megjelenítő egy tipikus képernyője (balról jobbra: végrehajtási nyom, statikus szelet, uniós szelet) 108
9.4. Uniós szeletek növekedése 109
9.5. Átlagos szeletméretek 110
9.6. A szeletek méreteinek eloszlása 111

Táblázatok jegyzéke

3.1. A teszt-rendszerek 24
3.2. Analizálási eredmények 24

8.1. A teszt-programok 85
8.2. A változók statisztikái 86
8.3. A D/U programreprezentáció méretei 86
8.4. Tesztesetek és statikus kritériumok 87
8.5. A szeletek méretei a *less* programhoz 88
8.6. Használt memória címek száma 90
8.7. A halmazműveletek statisztikái a *bzip* programhoz 91
8.8. Az igényvezérelt algoritmus futtatásai 94
8.9. Az iterációs szám statisztikái 97
8.10. A $worklist$ méretei 98

Tartalomjegyzék

Előszó **iii**

1. Bevezetés **1**
1.1. Eredmények összegzése 3
1.2. Felépítés 5

I. C/C++ forráskód analízis és modell-építés **7**

2. C/C++ forráskód analízis **9**
2.1. Forráskód analízis 10
2.2. C/C++ front end követelményei 10
2.3. Ismert megoldások 12
2.4. Összegzés 14

3. Analizálási technológia, megvalósítás **15**
3.1. Architektúra áttekintése 15
3.2. Elemzési technológia 18
3.3. Forráskód-modell előállítás 19
3.4. Speciális tulajdonságok 21
3.5. Tapasztalati eredmények 24
3.6. Összegzés 25

4. C++ analizátor alkalmazásai **27**
4.1. Columbus keretrendszer 27
4.2. További lehetséges alkalmazások 29
4.3. Összegzés 29

II. Dinamikus programszeletelés **31**

5. Programszeletelés **33**
5.1. Fogalmak 33
5.2. Dinamikus szeletelési algoritmusok 35
5.3. Egyéb szeletelési módszerek 37
5.4. Összegzés 37

6. Hatékony dinamikus szeletelési módszereink 39
6.1. Áttekintés . . . 39
6.2. Jelölések . . . 41
6.3. Statikus fázis . . . 41
6.4. Globális algoritmus . . . 43
6.4.1. Az algoritmus . . . 43
6.4.2. Komplexitás . . . 46
6.5. Igényvezérelt algoritmus . . . 47
6.5.1. Az algoritmus . . . 47
6.5.2. Komplexitás . . . 50
6.6. Összehasonlítás . . . 53
6.7. Összegzés . . . 54

7. Dinamikus szeletelési algoritmusok megvalósítása C nyelvre 55
7.1. C nyelvvel kapcsolatos problémák áttekintése . . . 56
7.2. Offline megvalósítás vázlata . . . 58
7.3. Statikus analízis . . . 58
7.4. D/U programreprezentáció C-re . . . 60
7.4.1. Adatfüggőségek . . . 62
7.4.2. Függvényhívások . . . 63
7.4.3. Vezérlési függőségek . . . 63
7.4.4. Összetett bal-értékek . . . 65
7.4.5. Mutatók, mutató indirekciók, *címe* operátor és tömbök . . . 66
7.4.6. Struktúrák és union-ok . . . 67
7.4.7. Típuskényszerítés . . . 68
7.5. Instrumentálás és a végrehajtási nyom . . . 69
7.5.1. A végrehajtási nyom fájlja . . . 69
7.5.2. Instrumentálás . . . 71
7.6. Globális algoritmus C-re . . . 73
7.6.1. Komplexitási tényezők . . . 74
7.7. Igényvezérelt algoritmus C-re . . . 75
7.7.1. Komplexitási tényezők . . . 77
7.8. Összegzés . . . 78

8. Szeletelési algoritmusok mérései 85
8.1. Teszt-programok és tesztesetek . . . 85
8.1.1. A D/U programreprezentáció méretei . . . 86
8.1.2. Tesztesetek és szeletelési kritériumok . . . 87
8.2. Szeletek méretei . . . 88
8.3. Számítási és tárterület komplexitás . . . 90
8.3.1. Használt memória címek száma . . . 90
8.3.2. Globális algoritmus . . . 91
8.3.3. Igényvezérelt algoritmus . . . 94
8.4. Összegzés . . . 100

9. Dinamikus szeletelési algoritmusok alkalmazásai 101
9.1. Nyomkövetés . . . 101
9.1.1. Releváns szeletek . . . 102
9.2. Uniós szeletek használata a szoftverkarbantartásban . . . 104
9.2.1. Realizálható és uniós szeletek . . . 105
9.2.2. Uniós szeletek számítása és kiértékelése . . . 106
9.2.3. Mérések . . . 107
9.2.4. Uniós szeletek alkalmazása . . . 110
9.3. Egyéb lehetséges alkalmazások . . . 112
9.4. Összegzés . . . 112

10. Konklúziók 113
10.1. Távlati tervek . . . 114
10.1.1. C/C++ analizátor és keretrendszer . . . 114
10.1.2. Dinamikus szeletelés . . . 115

Függelék 117

A. Magyar nyelvű összefoglaló 117
1. Bevezetés . . . 117
2. C/C++ forráskód analízis és modell-építés . . . 118
3. Dinamikus programszeletelés . . . 119
4. Konklúziók . . . 120

B. Summary in English 121
1 Introduction . . . 121
2 C/C++ Source Code Analysis and Model Creation . . . 122
3 Dynamic Program Slicing . . . 123
4 Conclusions . . . 124

Irodalomjegyzék 125

Bencének szeretettel
— Apa

1. fejezet

Bevezetés

„A számítógépek jók az utasításaink követésében, de rossz gondolatolvasók.”

— Donald E. Knuth

Közhelynek számít az a megállapítás, mely szerint a számítógép nagyon erős, de rendkívül ostoba. Ahhoz, hogy azt tegye, amit mondunk neki, szoftvereket kell kifejlesztenünk. A szoftverfejlesztés lényegében annyit takar, hogy az elképzelt szoftverről ember által megalkotott absztrakt igényeket, óhajtott funkcionalitást a számítógépnek is „elmagyarázzuk”, azaz egy olyan formában adjuk át, melyet végre tud hajtani: ez a gépi kód. A két véglet között a fejlesztési életciklus során számos, különböző absztrakciós szinteken lévő modellt származtatunk, amelyek mind leírják ugyanazt a szoftvert. E modellek közül a forráskód az, amely a végső kapcsolatot képviseli ember és gép között, hiszen azt általában a programozók írják az absztraktabb modellek alapján és a fordítóprogram segítségével alakítják át – teljesen automatikus módon – gépi kóddá. Tehát a magas szintű programozási nyelveket elsődlegesen a számítógépnek kell megértenie (pontosabban a fordítóprogramnak). A forráskód mérete és összetettsége ez utóbbinak nem jelent problémát, egyedül a rendelkezésre álló erőforrások képviselnek korlátot. Az is igaz, hogy a fordítóprogram *nem értelmezi* a kapott programot, csak mechanikusan elvégzi a szükséges átalakításokat. Ezzel szemben, számtalan ok lehet arra, hogy ember értelmezze a forrást, megértse annak működését, vagy egyéb szempontból következtetéseket vonjon le róla. A forráskód értelmezése nem könnyű feladat, főleg ha figyelembe vesszük a lehetséges komplexitást és méretet, valamint azt, hogy az ember számára a kód sokszor ismeretlen: más személy alkotása, vagy ha saját kódunkról van szó, a mögötte lévő ráció feledésbe merül.

A kód értelmezésének igénye az életciklus szinte bármely pontján felmerülhet, kezdve a nyomkövetés-hibajavítással, amikor a hibás viselkedés okát próbáljuk felfedni. A hibajavítás a kód első megjelenésekor felmerül, de a verifikáció fázisában, a program tesztelésének eredményeként is. Továbbá, a fejlesztés után, a szoftver evolúciós fázisában a karbantartás alapvető tevékenység, amikor sok esetben a forráskód az egyetlen megbízható reprezentációja a rendszernek. Ez adódhat abból, hogy manapság a szoftverrendszerek rendkívül gyorsan változnak, újabb és újabb verziók megjelenését követeli meg a piac és a rohamosan fejlődő új technológiák. Az új verziók sokszor a meglévő rendszerekből jönnek létre evolúció által, ami a gyors fejlesztés miatt legtöbbször azt eredményezi, hogy a kódot kísérő magasabb szintű modellek hiányosak, nincsenek összhangban a kóddal, vagy teljes egészében hiányoznak. Ezen problémák eredményezték az *újratervezés* (reengineering) tudományág létrejöttét a szoftverfejlesztésen belül. Ezen tevékenység részeként első lépésként mindig szükség van

a *meglévő rendszerek modellezésére* (reverse engineering), amikor is a rendelkezésre álló forráskódból megpróbáljuk meghatározni a rendszer komponenseit és az azok közötti összefüggéseket, továbbá előállítani egy rendszer magasabb absztrakciós szinten lévő, vagy más jellegű reprezentációját [20].

A fentiek miatt láthatjuk, hogy nagy igény van a forráskód megértését támogató eszközökre, hiszen a megértés folyamatában számos lépés mechanikus, amelyeket eszköztámogatással egyszerűbbé, gyorsabbá tehetünk. Mint ahogy a fordítóprogram is képes előállítani különböző reprezentációkat a programról, melyekre szüksége van saját céljának eléréséhez, úgy automatikus, kódot *analizáló* eszközök is kifejleszthetők, melyek egyéb reprezentációkat tudnak létrehozni. Az igazi haszon az, hogy a forráskód analízisével, különböző reprezentációk előállításával és egyéb kérdések megválaszolásával a forráskódról, ember által könnyebben felfogható formában tudjuk prezentálni a szoftverrendszert.

Az eszközzel támogatott programmegértés számos problémát vet fel. Ilyenek a rendszerről feltárt tények halmaza, azok reprezentációjának és tárolásának módja, valamint egyéb eszközökkel való kapcsolat megteremtése. A feltárt tényeket nevezzük egységesen a forráskód *modelljének*, az abban lévő adatok struktúráját pedig *sémának*. Az eszközök közötti kapcsolattartás alapja a közös séma és fizikai formátum. A másik alapvető eleme a megértést támogató eszközöknek az *analizátor*, amely a forráskód elemzésével és különböző összefüggések származtatásával képes a tények feltárására. Ezután maguk az eszközök számtalan módon lehetnek alkalmasak a megértés támogatására, amelynek egy példája a programszeletelés. Szeleteléskor a teljes program helyett annak csak egy részhalmazát nyújtjuk a felhasználónak, amelyen a kívánt vizsgálatokat kell elvégezni, és ezáltal csökkentjük a problémájának nagyságát.

A fenti problémakört tűztük ki feladatunknak, amikor belevágtunk egy olyan technológia- és eszközrendszer kidolgozásába, amelynek néhány alapvető eleme máris működőképes, nemzetközi elismerésnek örvend és valós környezetben használatos. E hosszútávú fejlesztésünk jelenlegi eredménye egy általános analizáló keretrendszer, melynek neve Columbus [30, 31, 32, 33, 34], és a szükséges technológiák és eszközök. Jelenleg a C/C++ nyelveket céloztuk meg, hiszen ezen nyelvek a legelterjedtebbek között vannak és már elég régóta léteznek ahhoz, hogy egyre több, bennük íródott rendszernél legyen szükség újratervezésre, megértésre. Fejlesztésünk konkrét eredményei egy általános analizátor és séma C/C++ nyelvekre, valamit egyéb, forráskód analízisen alapuló eszközök, mint amilyen a dinamikus szeletelőnk [8, 46, 12]. Kísérleteink végzéséhez szükséges eszköztárunkban szerepel még néhány külső eszköz is, amelyek területeivel pillanatnyilag nem kívánunk foglalkozni. Ezek közé sorolhatók a különböző szoftver-vizualizáló eszközök, és egy statikus szeletelő is.

Jelen értekezés szerzője a programmegértést támogató környezetünk két fontos elemét dolgozta ki. Az egyik a minden analizálási feladat alapja, egy általános célú C/C++ forráskód analizátor, amely meghatározott sémának megfelelő modellt állít elő. Ezzel foglalkozik a dolgozat első része. Mint látni fogjuk, ez a fordítóprogramok elemzőihez képest számos tekintetben speciális kellett hogy legyen. A másik eredmény, amelyet a dolgozat második részében ismertetünk, a dinamikus programszeletelő, egy speciális kód-analizálási eszköz, amely sokrétű alkalmazási területekkel bír. Utóbbihoz két hatékony algoritmus és C nyelvre való megvalósítás tartozik, és ellentétben a korábbi módszerekkel, valódi alkalmazásokban is használható hatékonysága és részletekbe menő kidolgozottsága miatt.

1.1. Eredmények összegzése

Az értekezésben ismertetett eredményeket négy tézisben határozzuk meg, amelyekből egy a dolgozat első témájához kapcsolódik és három a másodikhoz, az alábbiak szerint.

C/C++ forráskód analízis és modell-építés

A fordítóprogramokban és forráskód analizátorokban használt szintaktikus elemző modult, kiegészítve további feldolgozással, melynek során előállít egy belső reprezentációt, *front end*-nek hívjuk. Egy általános C/C++ analizátor front end számos területen eltér a fordítóprogramokban használatostól, hiszen mások a céljaik. A jelenleg elérhető front end-ek vagy megfizethetetlenek vagy valamilyen szempontból alkalmatlanok a céljainkra, például nem teljes, vagy hibás elemzést végeznek, esetleg nincs jól definiált kapcsolódási lehetőségük. Emiatt fejlesztettük ki az alábbit:

I/1. *C/C++ forráskód analízis és modell-építés.* A szerző által kifejlesztett analizátor front end technológiában és eszközben teljesítettük az általános analizátoroknak támasztott követelményeket, emellett az eszköz egy jól meghatározott sémának megfelelő modellt képes előállítani, és könnyű bővíthetőséget lehetővé tevő interfészei vannak. Egyedi megoldásokat alkalmaztunk többek között a hibatűrés, analizálási sebesség, dialektusok kezelésére.

Kapcsolódó publikációkban szereplő eredmények A [29] cikkben ismertettük a C++ sémát, amelynek megfelelő modellt épít fel a front end, sajátos technológiával (a séma nem, de az építés technológiája a szerző eredménye). A [34] publikációban mutattuk be az általános C++ front end-ekkel szemben támasztott követelményeket. Továbbá, ismertettük az analizátorunkat főbb tulajdonságaival és néhány mérési eredményt tettünk közzé, ezek a szerző munkái. A [101] közlésben ismertettük az előfeldolgozót és sémát, valamint néhány minta modellt (az előfeldolgozási séma nem a szerző munkája). A előfeldolgozóban használt építési stratégia és technológia a C++ nyelvi elemzőben alkalmazott alapján készült, amely a szerző eredménye.

Dinamikus programszeletelés

A dinamikus programszeletelés [3, 67] egy olyan analizálási technika, amely során egy program adott futására meghatározzuk annak egy (általában nem végrehajtható) részhalmazát, amely tartalmazza azon utasításokat, melyek közvetlenül vagy közvetetten kihatással voltak egy adott program-pont adott változó-előfordulásainak értékeire. A dinamikus szeleteket számos célra javasolják a szoftver-karbantartásban és programmegértésben, egyik alapvető alkalmazásuk a nyomkövetés. Sajnos a meglévő szeletelő algoritmusok alkalmazási köre korlátozott rossz hatékonyságuk miatt. Továbbá, a publikált módszerek nem szolgáltatnak kellő információt valós nyelvek dinamikus szeletelésének problémáira. Emiatt két szeletelési algoritmust dolgoztunk ki, amelyek valós méretű C nyelvű programok analízisére alkalmasak. A két módszer ugyanazon eredményeket szolgáltatja, de megvannak az előnyeik és hátrányaik egyaránt, így mások lehetnek a potenciális alkalmazásaik is. Az algoritmusokhoz elvégeztük a komplexitási vizsgálatokat, és kísérleteink révén részletes mérési eredménye-

ket is előállítottunk, amelyek alapján képesek voltunk összehasonlítani a két módszert. Az eredmények a következők:

II/1. *Globális dinamikus szeletelő algoritmus C nyelvre.* Szerzőtársaimmal együtt kidolgoztunk egy globális dinamikus szeletelési algoritmust, amelynek kiterjesztése valós procedurális programozási nyelvekre (C-re) a szerző munkája. Az algoritmus egy adott futás összes dinamikus szeletét meghatározza. A módszer minden részletre kiterjedő megoldást nyújt többek között az interprocedurális működésre, a tetszőleges vezérlésátadási szerkezetekre, mindenfajta adat közötti függésekre, és a futási, dinamikus információk összegyűjtésére. A statikus információk származtatására az ismertetett C/C++ front end-et használjuk. Az algoritmus hatékonyságát a komplexitás vizsgálatával és részletes mérésekkel ellenőriztük.

Kapcsolódó publikációkban szereplő eredmények A [46] cikkben jelentettük meg először a globális dinamikus szeletelő alap-algoritmust, amely nem a szerző eredménye. A részletek kidolgozása, a prototípus megtervezése, a mérések elvégzése a szerző munkája. A [12] publikáció ismerteti a C programok szeletelésére vonatkozó részleteket, amelyek kidolgozása a szerző munkája. A cikk elnyerte a bemutatást fogadó konferencia legjobb munkájáért járó díjat. A konferencia a „European Conference on Software Maintenance", a legnagyobb európai és világviszonylatban is az egyik legjelentősebb szoftverkarbantartási konferencia.

II/2. *Igényvezérelt dinamikus szeletelő algoritmus C nyelvre.* A szerző munkája egy igényvezérelt dinamikus szeletelési algoritmus kidolgozása és részleteinek megadása valós procedurális programozási nyelvekre (C-re). Az algoritmus ugyanazon statikus információkat használja fel, de a végrehajtás feldolgozását másképp végzi és igény szerinti szeletet számít ki. Az algoritmus hatékonyságát ugyancsak a komplexitás vizsgálatával és részletes mérésekkel ellenőriztük.

A módszer nemzetközi publikálása az értekezés írásának idejében előkészítés alatt áll.[1]

II/3. *Dinamikus szeletelés alkalmazásai.* A dinamikus programszeletelési módszereknek számos alkalmazása közül a nyomkövetésnél használatosak a releváns szeletek, amelyek számításának részleteit a szerző dolgozta ki. Másik alkalmazás az uniós szeletek használata a szoftverkarbantartásban, amelynek technológiája a szerző munkája. Továbbá, kiterjedt mérésekkel támasztjuk alá az uniós szeletek létjogosultságát.

Kapcsolódó publikációkban szereplő eredmények A releváns szeletek számítását a [46] cikkben tettük közzé, ahol az alap-algoritmus nem a szerző eredménye. Az algoritmus részleteinek kidolgozása, a prototípus megtervezése, a mérések elvégzése a szerző munkája. A [8] publikációban ismertettük az uniós szeletek motivációját és számításuk technológiáját, amely a szerző eredménye, egyetemben a mérések megtervezésével és kiértékelésével.

Az alábbi táblázatban összefoglaltuk a felsorolt eredményekhez kapcsolódó publikációkat és összerendeltük azokat az értekezés fejezetszámaival:

[1] E tézis a jelen munkának hozzávetőlegesen 10–15 oldalában van megadva.

Tézis	Publikációk	Fejezetek
I/1.	[29, 34, 101]	2., 3., 4.
II/1.	[12, 46]	6., 7., 8.
II/2.		6., 7., 8.
II/3.	[8, 46]	9.

1.2. Felépítés

Az értekezés az eredményeknek megfelelően két részből áll, amelyek felépítését az alábbiakban ismertetjük.

I. rész

Az első rész a forráskód analízissel és a modell előállítással foglalkozik három fejezetbe sorolva. A második fejezetben áttekintjük a forráskód-analízis problematikáját, különös tekintettel a C/C++ nyelvek elemzésére és reprezentációjára. Meghatározunk egy követelménylistát, amelyet egy általános analizátor front end-nek támogatnia kell, valamint áttekintjük a meglévő technológiákat és eszközöket. A harmadik fejezet témája az általunk megvalósított analizálási technológia és elkészült eszköz. Áttekintjük a szintaktikus elemzési technológiát, a modell előállítását, és a front end általános célú felhasználását lehetővé tevő speciális tulajdonságait, és végül némi mérési eredményt is bemutatunk.

Az első részt a negyedik fejezettel zárjuk, amelyben röviden áttekintjük a front end meglévő és tervezett alkalmazásait.

II. rész

A dolgozat második részében a dinamikus szeletelési módszerekkel foglalkozunk, az alábbiak szerint. Az ötödik fejezet bevezetést nyújt a programszeletelés témájába, külön kitérve a dinamikus szeletelésre és a meglévő algoritmusokra. A két dinamikus szeletelési algoritmusunkat először elvi szinten adjuk meg, egyszerű programok esetére, amelyet a hatodik fejezetben ismertetünk. E fejezet célja az algoritmusok alapelvét bemutatni, példával illusztrálva. Megadjuk az algoritmusok komplexitási vizsgálatát, és végül összehasonlítjuk a kettőt. A hetedik fejezetben kiegészítjük a két elvi algoritmusunkat C programok szeletelésére. Áttekintjük a felmerült problémákat és részletesen ismertetjük az alkalmazott megoldásokat. A szeletelőkkel végzett részletes méréseink eredményét a nyolcadik fejezetben tárgyaljuk, és végül a kilencedik fejezetben a szeletelési algoritmusok alkalmazásaival foglalkozunk.

A dolgozatot a konklúziókkal és a jövőre vonatkozó terveinkkel a tizedik fejezetben, valamint magyar és angol nyelvű összefoglalókkal a függelékben zárjuk.

I. rész

C/C++ forráskód analízis és modell-építés

2. fejezet

C/C++ forráskód analízis

„A fő programozási nyelvek közül a C++-ban fordul elő legtöbb szójáték és poén. Ez nem véletlen.''

— Bjarne Stroustrup

A C programozási nyelv volt az első (és talán egyetlen) nyelv, amely ötvözte az alacsony szintű (rendszer-) programozás hatékonyságát és korlátlanságát a magas szintű nyelvek absztrakciójával. Az alacsony szintű lehetőségek (mint amilyenek a közvetlen memória elérés és mutató aritmetika) miatt rendkívül hatékony nyelvként is nyilvántartják. A „nagyobbik testvér'', a C++ nyelv gyakorlatilag megtartva a C minden előnyét, még magasabb szintű programozást tesz lehetővé az objektumorientáltság támogatásával, ezért az utóbbi években az egyik legelterjedtebb nyelvként ismerjük (a C++ gyakorlatilag tartalmazza a C nyelvet, ezért a továbbiakban a két nyelvet együtt tárgyaljuk). Ehhez még hozzáadódik a rendkívül jól kidolgozott standard könyvtár, és így egy hatékony, ugyanakkor magas absztrakciót is nyújtó nyelvvel állunk szemben. E kettősségnek természetesen ára van, ami két tekintetben is jelentkezik. Az első a lecsökkent biztonság (szabad memóriagazdálkodás, típusosság hiánya), legalábbis összehasonlítva néhány újabb programozási nyelvvel, mint például a Jávával. A másik velejáró a nyelv bonyolultsága, ami azt is jelenti, hogy jó C/C++ programozóvá (aki ki tudja használni a hatékonysági előnyöket, ugyanakkor biztonságos kódot készít) csak sok-sok tapasztalat útján válhatunk.

De a mi szempontunkból nem ez a legfontosabb tényező, hanem az, hogy a nyelvek bonyolultsága azok korrekt feldolgozása szempontjából is komoly kihívást jelent. Vannak akik a C++ nyelvet az emberiség egyik legkomplexebb szellemi alkotásának tekintik, ezért a nyelvet teljes egészében támogató fordítóprogram vagy egyéb forráskód analizátor is szükségszerűen rendkívül bonyolult lesz. A komplexitás adódik egyrészt a bonyolult, kétértelműségekkel teli szintaxisból (ami már a C-nél is jelentkezik). Másrészt, – ami még súlyosabb – számtalan további szemantikai szabály és egyéb előírás, javaslat alkotja a nyelv pontos meghatározását (a C++ nyelv ISO és IEC szabványként van rögzítve [56]). Maga Stroustrup, a nyelv megalkotója is elismeri a C++ komplexitását [95], de ez a komplexitás szükségszerűen adódik a nyelv eredeti tervezési elveiből.

Jelen fejezetben áttekintjük, hogy miért nem alkalmasak a fordítóprogramokban alkalmazott analizálási technológiák egyéb forráskód-analízis célra, és hogy miért kellett saját C/C++ analizátort kidolgoznunk céljaink eléréséhez.

2.1. Forráskód analízis

Mint ahogy egy fordítóprogramnak a nyelv szabályainak megfelelő forrást kell elemeznie és megfelelő módon értelmeznie, ugyanúgy egy egyéb célú analizátornak is hasonlóan kell eljárnia. Számos egyéb célt tudunk megfogalmazni a szoftverkarbantartás, a meglévő rendszerek utólagos modellezése (reverse engineering), dokumentálása, megértése és validálási alkalmazásokban. Bármely forráskódot feldolgozó eszköznek van egy forráskódot analizáló modulja, amely az adott nyelv szintaktikai és egyes szemantikai szabályainak megfelelően elemzi a forráskódot, és abból előállít egy magasabb szintű reprezentációt a további feldolgozás részére (például absztrakt szintaxis fát). E modul közismert elnevezése a *front end*, vagyis elülső rész, ahol a feldolgozás kezdődik. Az, hogy a front end feladata hol végződik, és mikor kezdődik az analízis következő fázisa, a feladattól függ és nincs rá egységes definíció. A határ lehet a fa, vagy mélyebb analízis eredménye, például folyamgráf. Az analizátorokban lévő front end-et szokták még tény-feltárónak (fact extractor) is nevezni.

A fordítóprogramokban és más analizátorokban lévő front end-eknek legtöbbször különbözőek a céljaik, ezért a fordítóprogramoknál alkalmazott módszerek nem mindig felelnek meg egyéb célokra is. Például, a fordítóprogram a forráskód elemzésével előállítja a program absztrakt szintaxis fáját, amibe a további fordításhoz szükségtelen információk már nem kerülnek bele, ezért végleg elvesznek. Ilyenek például a kommentek, vagy akár a programban szereplő, szintaxist elősegítő, vagy redundáns utasítások, deklarációk. Program karbantartás, megértés, dokumentálás – azaz tény-feltárás – céljára ezek pedig igenis fontosak lehetnek. Egy másik nehézség a fordítóprogramok front end-jével az, hogy azok általában csak nagyon kis mértékben hibatűrők, tehát ha hibát észlelnek a programban, nem tesznek különösebb erőfeszítéseket, hogy a hiba utáni részekből összegyűjtsék az összegyűjthető információkat, hiszen a fordítás már úgyis sikertelen lesz. Ugyanakkor hiányos, vagy hibás forráskódból is kinyerhető hasznos információ sok alkalmazásban.

A fentiek miatt egy általános programmegértést és karbantartást támogató analizátorban speciális front end-et kell alkalmazni, amelynek sajátos tulajdonságokkal hogy rendelkeznie. A másik megközelítés szerint egyedi forráskód feldolgozási feladatokra egyedi analizátorokat készítünk, viszont ezek kevésbé, vagy egyáltalán nem lesznek újrafelhasználhatók egyéb alkalmazásokban. A mi céljaink kielégítésére úgy döntöttünk, hogy egy általános C/C++ front end-et fejlesztünk ki, jól definiált kapcsolódási lehetőségekkel, hogy minél szélesebb körben felhasználhassuk.

A következő alfejezetekben áttekintjük, hogy egy általános front end-nek milyen követelményeknek kell eleget tennie, illetve megnézzük, hogy az elérhető egyéb technológiáknak milyen hátrányaik vannak, amik miatt új megoldást kellett találnunk.

2.2. C/C++ front end követelményei

Túl a nyelv helyes felismerésén, egy általános célú analizátor front end-jének számos olyan követelménynek kell eleget tennie, amelyek a fordítóprogramok esetében nem mindig játszanak szerepet. Ezért a fordítóprogramok által használt technológiák általában nem alkalmasak céljainkra. A [34] cikkben közöltünk egy követelménylistát, amelyet itt ismertetünk kiegészítve néhány további szemponttal:

- *Teljesség*. Az általánosság miatt a forráskód teljes analízisére van szükség, a nyelv által definiált összes információt ki kell nyerni. Nem elegendő az ún. *fuzzy* elemzési technika, amit számos tény-feltáró analizátor alkalmaz, hiszen ezek részleges reprezentációjára képesek csak a nyelvnek. Ugyanakkor, kényelmi szempont, ha a front end-ben választási lehetőség van a kinyert információ tartalmát illetően, hiszen ha egy alkalmazás kevesebb információval is beéri, a részletes elemzéstől eltekintve gyorsabb feldolgozás lehetséges.

- *Részletesség*. A szintaktikai elemzők által előállított magas szintű reprezentáció általában az absztrakt szintaxis fa, amelyben számos, szintaxist segítő nyelvi elem kihagyásra kerül. A fordítóprogramokban alkalmazott elemzők tipikusan kihagyják az olyan információkat mint pl. vesszők, zárójelek, kommentek, redundáns deklarációk, stb. Ezen kívül egyes forráskódbeli szerkezeteket ekvivalens formájúra alakíthatnak, például az `a!=b` kifejezésből `!(a==b)` lesz. Néhány tény-feltáró alkalmazásban azonban az eredeti, forrás-hű információkra van szükség, ezért a front end gyakorlatilag a forráskódban található összes információt ki kell hogy gyűjtse. A mi front end-ünkben ezt úgy oldottuk meg, hogy az alacsony szintű információkat az előfeldolgozó gyűjti ki, az elemző pedig nem dob el fontos elemeket.

- *Kapcsolódás*. Az elemzett forrásból előállított belső reprezentációban szereplő információ tetszőleges célra lesz felhasználva, ezért fel kell készülni az analizátor további részeivel való minél könnyebb kapcsolódásra. Ezért az információt egy olyan struktúrában adjuk tovább, amely általános célra felhasználható és az összes szükséges információt tartalmazza. Ezt az adathalmazt nevezzük a forráskód *modelljének*, azt a leírást pedig, amely megmondja, hogy a modellben szereplő adatok mik lehetnek és azok hogyan viszonyulnak egymáshoz, *sémának* fogjuk hívni. A keretrendszerünkben kidolgoztunk egy sémát, amely szerint ez az átadás történik. (ld. a következő fejezetet).

- *Front end határa*. A front end által előállított modell információ tartalma változó, ezt általában előre rögzíteni kell. A fent említett teljesség és részletesség meghatározza a minimális információ tartalmat. Kérdés, hogy a minimális modellen felül, az abból kiszámítható további adatokat milyen mértékben kell egy front end-nek előállítania. C++ esetében erre az első példa az olyan nyelvi elemek generálása, amelyek a forráskódban nem szerepelnek, de egy helyes programhoz implicite létezniük kell (ilyen például az implicit konstruktor). Ezen kívül további hasznos származtatott információkat is előre el lehet készíteni, mint amilyen a folyamgráf, vagy a program-függőségi gráf. A mi front end-ünk az absztrakt szintaxis szintjén marad.

- *Hibatűrés*. Tény-feltáráskor sokszor szembesülünk olyan szituációkkal, amikor az analizálandó forráskód nem teljes (például mert még fejlesztés alatt áll, vagy nem áll rendelkezésre a teljes forráskód bázis), esetleg hibás (nem fordítható). Egy általános célú front end-nek képesnek kell lennie a lehető legtöbb kinyerhető információt összegyűjteni. Tehát a fordítóprogramhoz képest jobb hibatűrési képességekkel kell rendelkeznie.

- *Sebesség*. A sebesség kevésbé kritikus egy analizátornál, mint a fordítóprogram esetében, hiszen az utóbbit valószínűleg gyakrabban futtatjuk, főleg fejlesztési fázisban.

Ugyanakkor, ne feledjük el, hogy egy karbantartási célt szolgáló analizátornak általában nagy mennyiségű forráskódot kell feldolgoznia egyhuzamban, tehát a sebesség kérdése nem teljesen elhanyagolható. A mi front end-ünkbe beépítettünk néhány sebességnövelő szolgáltatást.

- *Előfeldolgozás*. C/C++ front end esetében az előfeldolgozást is nyilvánvalóan el kell végezni (például a makrók kifejtését vagy a fejlécfájlok feldolgozását). Ezen kívül az előfeldolgozóval kapcsolatos információkat is ki kell gyűjteni, hiszen azokra is szükség lehet (például *include*-hierarchia előállítása). A mi front end-ünkben ezt úgy oldottuk meg, hogy különálló előfeldolgozót és nyelvi elemzőt készítettünk, amelyekhez saját sémák is tartoznak.

- *Nyelvi dialektusok*. A C és C++ nyelveknek számos dialektusa ismert, főleg az egyes fordítóprogramok által meghatározott nyelvi bővítésekből adódóan. Mivel egy általános front end-et nem köthetünk egy adott fordítóprogramhoz, a lehető legtöbb használatban lévő dialektust ismernie kell. Továbbá, az ismeretlen dialektusok kezelésére biztosítani kell valamilyen bővítési mechanizmust a felismert nyelvet illetően.

- *Felhasználói felület*. Nem elhanyagolható az a kérdés sem, hogy a front end-et, illetve az analizátort milyen módon lehet beilleszteni az őt használó meglévő környezetbe. A legáltalánosabb használhatóságot az jelenti, ha az eszköz parancssori üzemmódban (is) tud működni, hiszen így tetszőleges, akár grafikus felhasználói felület segítségével behívható. Továbbá, a legtöbb analizálandó valós szoftverrendszert valószínűleg valamilyen automatikus fordítási környezet kísér, amelybe a legkönnyebb beépülést a parancssori működés biztosítja.

2.3. Ismert megoldások

Az egyéb, tény-feltárásra is használt, C++ front end-ek között vannak olyanok, amelyek általános célúak (esetleg fordítóprogramokban is használatosak), míg mások valamilyen speciális feladat-halmaz elvégzésére alkalmasak. Az alábbiakban röviden áttekintjük a nyilvánosan elérhető analizátorok közül a legfontosabbakat, és megnézzük, hogy az előző alfejezet követelményeivel milyen viszonyban állnak.

Talán a világ egyik legismertebb front end szállító cége az Edison Design Group [28]. Az *EDG* C/C++ front end széles körben arról nevezetes, hogy rendkívül pontos, a szabványt nagymértékben támogató elemzővel rendelkezik. Számtalan kereskedelmi fordítóprogram ezen front end-et használja, főleg beágyazott rendszerekhez különböző célprocesszor támogatással. Annak ellenére, hogy a fő alkalmazási területe a fordítóprogramoknál van, néhány analizátorról is tudunk, amelyek használják ezt a front end-et. Sajnos a front end-nek és az azt használó termékeknek borsos ára van, ezért nem állt módunkban mélységeiben megismerni a technológiát. Az általunk elkészített front end bárki számára ingyenesen elérhető kutatási célra.

Hasonló jellegű kereskedelmi termék a Semantic Designs cég C++ front end-je is [91]. Azonban a front end újrafelhasználása sokkal flexibilisebb, hiszen itt a nyelvtani leíráshoz lehet közvetlenül kapcsolódni, és szükség szerint módosítani azt (GLR-alapú – általánosí-

tott LR – elemzési technológiát használnak, amellyel tetszőleges környezetfüggetlen nyelv elemezhető [98]).

A különálló front end-eken kívül még számos integrált fejlesztőkörnyezetben is található forráskód-analizáló modul. Ezek azonban általában csak az adott környezet céljait szolgálják és nem használhatók általános célra. Ilyenek például a *Rational Rose* [85] vagy a *Together* [15].

Míg a kereskedelmi termékek céljainkra való felhasználásának fő gátló tényezője a magas áruk, addig az ingyenes eszközök és technológiák egyike sem felelt meg maradéktalanul az előző alfejezetben ismertetett követelményeknek. Az alábbiakban áttekintjük az ismertebb megvalósításokat.

A *Datrix* analizátor [76] saját séma szerinti modellt állít elő C, C++ és Jáva forrásokból. A *Datrix* hasonló elveken alapul, mint a mi analizátorunk és sémánk, de mivel a kutatási projekt idő előtt leállt, az eszközök is befejezetlenek maradtak.

Tipikus fordítóprogram alapú analizátor a *CPPX* [24], amely ingyenes, nyílt forráskódú általános C++ tény-feltáró. Alapja a GNU GCC (g++) fordítóprogram, és emiatt a fent említett hátrányok mind terhelik. Kimenetét különböző fájlformátumokban állítja elő a Datrix séma alapján. Ugyancsak GCC alapúak, de kevésbé jelentős projektek az *XOgastan*, a *gasta* és a *gccXfront*. Ezen kívül a *Cetus* Jáva-alapú fordítóprogram-keretrendszer [17, 72] is tartalmaz C++ front end-et, amely ugyancsak alkalmatlan céljainkra (nem általános, és még nem teljesen kiforrott).

Az *OpenC++* [19, 80] is egy fordítóprogramon alapuló eszközrendszer, amely segítségével a C++ forráskód-analizálási, -transzformáció és -nyelvi átírási feladatokat magas szintű leírásokkal adhatjuk meg. Ezután magát az analizátort automatikusan állíthatjuk elő a rendszer segítségével. Ez merőben más megközelítése a forráskód-analizálási problémának és sok esetben nagyon hatékony analizátor-fejlesztést tehet lehetővé. Az analizálási feladatunkat az ún. *metaobjektum-protokoll* segítségével adhatjuk meg, ami gyakorlatilag egy C++ nyelvű programozási interfész. Az ilyen meta-programot lefordítjuk az OpenC++ segítségével, és hozzászerkesztjük a fordítóprogramhoz. Ezután, az így kiegészített fordítóprogrammal végezzük el a tényleges analízist. A technológia ugyanazon hátrányokkal rendelkezik, mint az összes fordítóprogram alapú megoldás, mint például a elemzési részletek hiánya, rossz hibatűrés, nyelvi dialektusok hiánya, stb.

A Red Hat által karbantartott *Source-Navigator* nevű eszköz [93] egy grafikus felhasználói felülettel rendelkező forráskód-analizáló eszköz. A C++ mellett egyéb nyelveket is támogat, de az alkalmazott analizálási technológia nem általános célú, hanem csak az eszköz által definiált különböző lekérdezések támogatására alkalmas. Az elemző egy fuzzy-elemző, amely nagyon gyors ugyan, de nem végez teljes elemzést, csak a szükséges, általában magasabb szintű információk kinyerését végzi (például osztály-öröklődési hierarchia). A forráskód elemzése után az eszköz a feltárt információkat egy adatbázisban tárolja, amit a grafikus lekérdezések és böngészések során használ fel. Ezen kívül további hiányosságai is vannak a front end-nek: nincs parancssori működés, nincs jól definiált adat-séma, stb.

Hasonló okoknál fogva a *SNiFF+* eszköz [92] sem alkalmas általános célú analizátorként való használatra. Az alkalmazott elemző itt is egy fuzzy-elemző, amely csak adott mennyiségű információt képes kigyűjteni a forráskódból.

A *rigi* eszköz [78, 86] egy általános gráf vizualizáló eszköz, amely tartalmaz C/C++ forráskód-analizáló modulokat is, azonban hasonlóan a korábbiakhoz, ez is csak részleges

elemzést végez, ezért nem tekinthető valódi alternatívának. C++ elemzéséhez a Source-Navigator eszköz elemzőjét használják.

További C++ analizátorok, front end-ek, vagy csak részleges megoldások (például nyelvtan) az alábbiak. A *Keystone* analizátort [74, 84] meglévő rendszerek modellezésére fejlesztették ki, amely a C++ nyelv elemzésével kapcsolatos néhány nehézség kezelésére fekteti a hangsúlyt (pl. név-hivatkozások feloldása), ugyanakkor egy általános front end követelményeit nem teljesíti maradéktalanul. A [104] cikkben ismertetett front end viszonylag régi, és a nyelv számos problémáját felületesen kezeli. A *sage++*, valamint Jim Roskind és John Lilley nyelvtanai/elemzői, továbbá az ANTLR elemzőgenerátor „hivatalos" C++ elemzője (utóbbi az [5] címen megtalálható) a C++-t részlegesen kezelik, valamint a szintaktikus felismerésen túl nem sok információt szolgáltatnak.

Újszerű megközelítése van a *Spirit* elemzőgenerátornak [94], amelyben meta-programozási technikával (C++ sablonok segítségével) adhatjuk meg a környezetfüggetlen nyelvtant az elemzendő nyelvhez. A projekt honlapján megtalálható egy C++ nyelvtan is, de ez még távol áll egy teljes analizátor front end-től.

2.4. Összegzés

Az előző alfejezetben számos C/C++ front end megoldást láttunk, azonban ezek vagy elérhetetlenek kutatási célra, vagy kapcsolódási felületük nem kielégítő, vagy pedig nem teljesítik kielégítően a 2.2. alfejezetben leírt követelményeket. A fentiek miatt saját C/C++ front end-et fejlesztettünk ki, amelyet az alkalmazott megoldásokkal együtt a következő fejezetben részletesen ismertetünk.

3. fejezet

Analizálási technológia, megvalósítás

„Nincs olyan általános szabály, ami alól ne lenne valamilyen kivétel.”

— Robert Burton

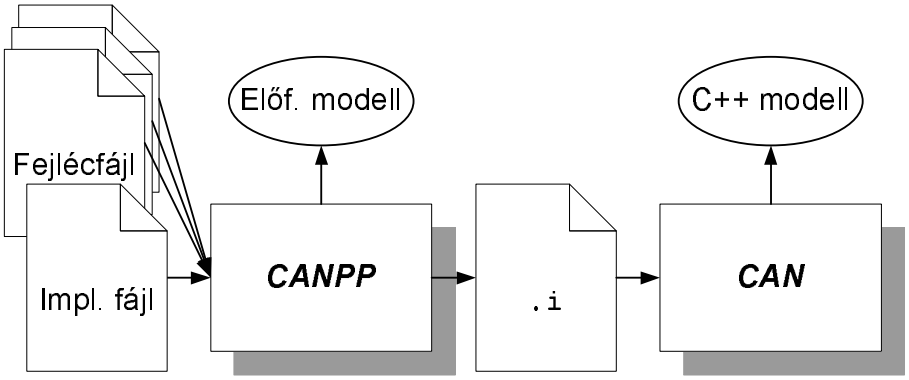
Az előző fejezetben leírtak alapján meggyőződhettünk róla, hogy a C (de méginkább a C++) analízise rendkívül nehéz feladat. Nem elegendő egy nyelvtan megadása – ami önmagában sem könnyű, hiszen a nyelvhez nem adható meg a hatékony elemzési technológiákkal ($LL(1)$ vagy $LALR(1)$) elemezhető nyelvtan. Egy teljes front end-nek meg kell oldania a felismert forrás megfelelő belső reprezentációját, valamint annak továbbadását az analizáló eszközök további részeinek. Ezen felül az előző fejezetben ismertetett egyéb funkcionális és nem funkcionális követelményeknek is eleget kell tennie.

Ebben a fejezetben bemutatjuk a mi kísérletünket az említett nehézségek megoldására, azaz az általunk alkalmazott technológiákat és a megvalósult C/C++ front end-et. Célunk az volt, hogy a létező rendszerek modellezésében, a szoftver-karbantartásban, programmegértésben, verifikációban, stb. érdekelt kutatóknak ne kelljen a forráskód analízisével foglalkozniuk, szolgálja ki a mi eszközünk a kutatási területeiknél szükséges forráskód-analízis igényét. A front end hasznát számos alkalmazásban magunk is bizonyítottuk, beleértve ide a programszeletelést is, amely jelen értekezés második részének témája. Az alkalmazásokról a következő fejezetben részletesen is szólunk.

3.1. Architektúra áttekintése

Az analízis teljességét és részletességét (ld. előző fejezetet) úgy oldottuk meg, hogy a front end-ünkben két eszközt használunk. Hasonlóan a C/C++ fordítóprogramokhoz, különálló előfeldolgozó (preprocessor) végzi el a neki szánt feladatokat, miközben kigyűjti a szükséges információkat a modell előállításához is. Az előfeldolgozási elemektől mentes forrás további feldolgozását a fő nyelvi feldolgozó végzi, amely ugyancsak előállítja a megfelelő modellt. A 3.1. ábrán láthatjuk a teljes front end átfogó felépítését, a *CANPP* nevű előfeldolgozóval és a *CAN* C/C++ elemzővel (em C++ Analyzer és *CAN Preprocessor*).

Láthatjuk, hogy az előfeldolgozó elvégzi a bemeneti forrásfájlok szabályos feldolgozását, előállítva az előfeldolgozott forrást (`.i` fájl, vagy belsőleg átadott adatfolyam), amely azonos a fordítók által előállított forrással. Emellett kiadja az előfeldolgozó sémának megfelelő modellt is, amelyet a további analízisekhez lehet felhasználni. A sémát a [101] munkában ismertettük, és az tartalmaz minden részletes információt a forrás alacsony szintű elemeiről,



3.1. ábra. A C/C++ front end felépítése

úgy mint az egyes *tokenek* (ezek lexikális szintű egységek, amelyek gyakorlatilag a nyelvi szintaxis betűit képviselik) és egyéb előfeldolgozási információk (például makrók és helyettesítéseik). A következő lépésben a CAN nyelvi elemző dolgozza fel az előfeldolgozott forrást és állítja elő a megfelelő nyelvi forráskód modellt. A két modell együttesen képviseli az eredeti forrásfájlból kigyűjtött összes információt (ld. 3.3 alfejezetet).

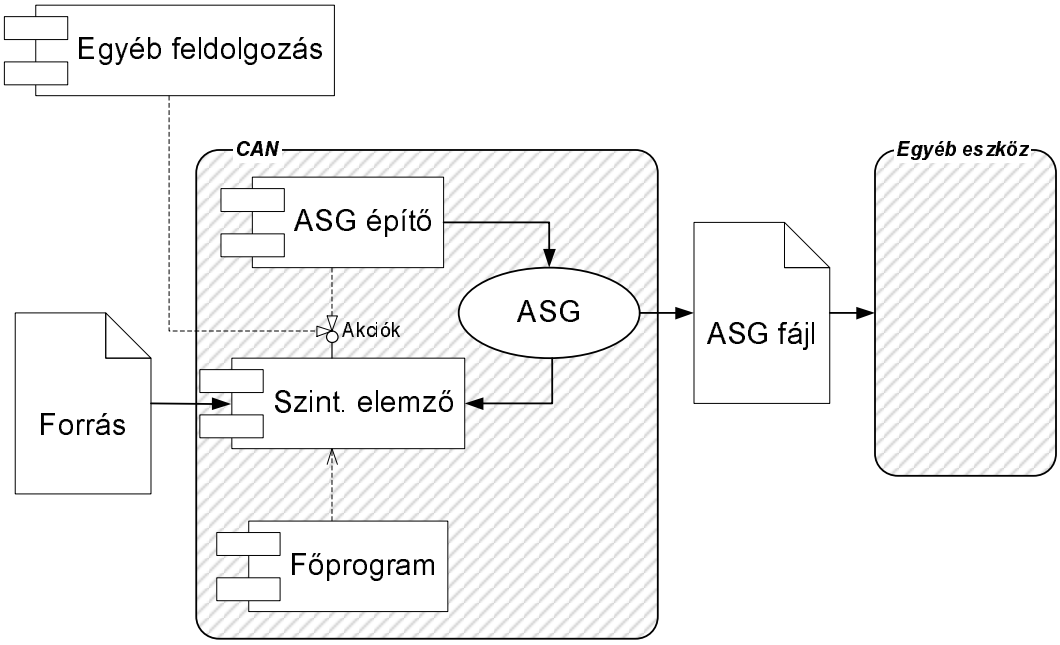
Jelen dolgozatban a CAN C/C++ elemzővel foglalkozunk, amely magában foglalja a szintaktikus elemzőt és a megfelelő modell előállítását végző mechanizmust.

Az elemző egy parancssori alkalmazás, amely implementációja szempontjából platformfüggetlen C++, tehát gyakorlatilag bármely operációs rendszerre lefordítható bármely alkalmas fordítóprogrammal. Ez elősegíti a legáltalánosabb felhasználását az eszköz futtatásával bármely analizáló rendszerből. Ilyen felállás szerint a feltárt információk átadását a sémának megfelelő fájlba kiírt adatokkal végezzük, amely feldolgozására rendelkezésre áll egy programozási interfész (API) is.

Ezen kívül az eszköz egy másik, még több lehetőséget kínáló interfészt is szolgáltat az elemző újrafelhasználására, amelyről később részletesen is szólunk. Ezen lehetőség viszont már a rendszer komponens szintű újrafelhasználását jelenti egy specializált eszközben. Emiatt az elemző felépítése moduláris, architektúrája a 3.2. ábrán látható.

A program bemenete egy teljes fordítási egység, előfeldolgozott forrás formájában. Ez azt jelenti, hogy az összes használt fejlécfájl tartalma elérhető, valamint a különböző előfeldolgozó elemek már le vannak kezelve (makrók, feltételes fordítás, stb.). A bemeneti forrást a szintaktikus elemző fogadja és ellenőrzi a nyelvi megfelelőségét. Az elemző egy felülről lefelé haladó, rekurzív alászálló eljárásokkal megvalósított $LL(k)$ alapú elemző kiegészítve számos, az elemzést elősegítő technikával. Az elemzőt környezetfüggetlen nyelvtani leírásból készítjük egy elemzőgenerátor segítségével. A nyelvtannal a szabványos C és C++ nyelveket, valamint néhány dialektust fedünk le. További részleteket a szintaktikus elemzővel kapcsolatban a következő alfejezetben adunk meg.

A forrásnak megfelelő belső reprezentáció elemzés közben készül, a szintaxis által vezérelve. Ez azt jelenti, hogy ahogy az elemző egy nyelvi elemet feldolgoz, közben ún. *szintaktikus akciókat* hajt végre. Az akciók gyakorlatilag egy interfészt képeznek, amely a szintaxisnak megfelelő operációkat definiálja. Az interfészt megvalósítva, az elemző által hívott operációk használhatók fel ezután a „hasznos munka" elvégzésére, esetünkben a belső reprezentáció és egyben a forráskód-modell elkészítésére, a gráf-építő modul tehát az akció-interfész egy megvalósítása. A modell gyakorlatilag egy absztrakt szintaxis gráf



3.2. ábra. A C/C++ elemző architektúrája

(ASG), amely megfelel a keretrendszerünk által definiált sémának, és ez nem más, mint az absztrakt szintaxis fa megfelelője kiegészítve a csomópontok közötti különböző relációkkal. Ilyenek például a feloldott név-hivatkozások, de részletesebb analíziseket, például folyamgráf számítást már nem végzünk a gráfon. A felépült gráf az elemzés végén kiíródik fájlba is, amely biztosítja az elemzőhöz való kapcsolódás alapvető módját. A gráf-építéssel a 3.3. alfejezetben foglalkozunk behatóbban.

Ugyanazon akció-interfészt felhasználva nyitott a lehetőség más, elemzés közbeni tevékenység elvégzésére is, amely újrafelhasználási lehetőségre fent utaltunk. A konkrét technikát a következő alfejezetben írjuk le.

A C++ nyelv korrekt elemzéséhez szükség van a szimbólumokról különböző strukturált információk eltárolására. Például az egyes azonosítók jelentését rögzíteni kell, amely a szintaktikus környezetből adódik (típus, változó, stb.), hiszen az elemzés szintaktikailag kétértelmű szituációit csak ezen információkat felhasználva lehet feloldani. Kézenfekvő ezen információkat is az ASG-ből kinyerni. Ez szerencsére megtehető az éppen építés alatt lévő ASG-t felhasználva, hiszen a nyelv szerint minden szimbólumnak deklaráltnak kell lennie a használat előtt (eltekintve egy kivételtől, amit a következő alfejezetben tárgyalunk). Így az ASG kettős célt szolgál: az elemző elősegítését és a modell tárolását.

A CAN programnak természetesen része még egy főprogram, valamint számos további segéd-komponens, amelyeket külön nem tüntettünk fel a fenti ábrán. A főprogram végzi a parancssori paraméterek értelmezését, a különböző bemeneti, kimeneti és ideiglenes fájlok kezelését, valamint a feldolgozó modulok felparaméterezését és behívását.

Az imént leírt CAN elemző a bevezetésben említett Columbus keretrendszeren belül működve lesz igazán hasznos, ugyanis az elemző által előállított modell egy fordítási egységnek felel meg, egy valódi feladatnál viszont általában a teljes rendszerre vagyunk kíváncsiak. A Columbus rendszeren belül egy teljes rendszer analízisekor a különállóan előállított modelleket összefűzzük úgy, hogy az ismétlődő entitásokat egyszer hagyjuk meg (*linkelés*), majd ezen összefűzött modellen – amely már az egész rendszert reprezentálja – végezzük el a

további feldolgozást (részleteket ld. például a [33, 34] cikkekben).

3.2. Elemzési technológia

Az elemzőnk hagyományos felépítésű: a bemeneti szöveget először tokenizáljuk, azaz elvégezzük a lexikális elemzést, amely előállítja a szintaktikus elemző számára a token-folyamot, mint bemeneti szimbólumsorozatot. A lexikális elemzés viszonylag könnyen elvégezhető determinisztikus véges automatával, hiszen az erre vonatkozó nyelvi definíciót reguláris kifejezésekkel is megadhatjuk. A lexikális elemző végez el még néhány további alacsony szintű feldolgozást, mint például a kommentek, nem nyomtatható üres karakterek, stb. felismerését.

Másfelől, a C++, nyelvi szintaxisát tekintve, nem könnyen elemezhető nyelv. A C nyelvhez még viszonylag könnyű hatékony elemzőt készíteni (1-2 konfliktus mellett $LALR(1)$ nyelvtannal elemezhető). A C++ szintaxisában viszont már olyan kétértelműségek vannak, amelyek miatt esetenként az egész deklarációt vagy utasítást meg kell vizsgálni, hogy kiderüljön milyen nyelvi elemről van is szó. Erre egy jó példa a `T(a)` kezdetű token-sorozat, amely lehet deklaráció vagy kifejezés prefixe is, amely tetszőlegesen hosszú lehet (a `T` azonosító típust nevez meg). További nehézségeket okoz a deklarációk, kifejezések és sablonok bonyolult szintaxisa.

A C++ elemzésére különféle megoldások születtek (ld. előző fejezetet), de szinte egyik esetben sem használnak tiszta elemzési stratégiát. Általában valamilyen kiegészítő, az alapstratégiát segítő megoldásokkal teli, sokszor kézzel írott elemző a végeredmény. A nyelv komplexitásának kezelésére egyes kutatók bonyolult és költséges technológiákat javasolnak, mint amilyen a természetes nyelvek elemzésére kidolgozott GLR stratégiának különböző fajtái (ld. [98]), vagy az ASF+DSF formalizmus [62]. A gyakorlatban azonban a környezetfüggetlen nyelvek elemzésére szolgáló hagyományos hatékony algoritmusok valamelyikének továbbfejlesztett változata terjedt el. A felülről lefelé haladó és alulról felfelé haladó algoritmusok közül az előbbi tűnik alkalmasabbnak a C++ elemzésére, és számos jelenlegi fordítóprogram is ilyen stratégiát alkalmaz (pl. GNU g++), legtöbb esetben rekurzív rászálló eljárásokkal megvalósítva. Ennek egyik oka talán az, hogy már a nyelv megalkotásakor Stroustrup e stratégia felé hajlott (ld. [95] 69. oldalát).

A fentiek miatt a mi elemzőnkben alkalmazott technológia is felülről lefelé haladó, erősen $LL(k)$-alapú algoritmus, $k = 2$-vel. Az algoritmus rekurzív eljárásokon alapul, ahol az eljárások egy-egy szabályt valósítanak meg. Adott szabály jobb-oldalainak megfelel a hozzá tartozó eljárás törzse, ahol az alternatíva eldöntését az erősen $LL(k)$ feltétel biztosítja. A jobb-oldalon szereplő nemterminális az adott eljárás meghívását, a terminális pedig a bemenet léptetését és a következő szimbólum beolvasását jelenti, a szimbólumok sorrendjének megfelelően (bal levezetés). Az elemzőt a nyelvtani leírásból automatikusan generáltatjuk a PCCTS rendszer segítségével [83], amely előállítja mind a lexikális, mind a szintaktikus elemző C++ forráskódját. Az elemzőgenerátornak EBNF formában [55] lehet megadni a szintaktikus nyelvtant, ami által egy áttekinthető, jól karbantartható rendszert kaptunk. (A lexikális elemzőre vonatkozó szabályok reguláris kifejezésekkel adottak.) A nyelvtan a C++ szabvány szerinti nyelvet ismeri fel [56], kiegészítve a C-re vonatkozó kisebb kiegészítésekkel és néhány fontosabb C++ dialektus bővítéseivel (ezekről a következő alfejezetben szólunk).

A PCCTS támogatja az ún. *predikátumos* $LL(k)$ elemzést, amely az elemzési nehéz-

ségek feloldására rendkívül hatékony módszer (ld. [82]). Ez ötvözi a kézzel írott, elemzést segítő/vezérlő megoldások hatékonyságát az elemzőgenerátor használatának kényelmével. A módszer alapja az ún. *szintaktikus* és *szemantikus predikátumok* használata, amelyek eldöntik az elemzés további irányát az olyan alternatíváknál, ahol az erősen $LL(k)$ feltétel nem teljesül és emiatt kétértelmű lenne a nyelvtan. A szintaktikus predikátum egy jobb-oldalhoz rendel tetszőleges szimbólumsorozatot, amelyet le kell elemezni úgy, mintha az egy valódi jobb-oldal lenne, és sikeres elemzés esetén az adott jobb-oldalt kell választani, egyébként a következő alternatívát kell megvizsgálni. A levezetett szintaktikus predikátum tipikusan az adott jobb-oldal levezetésének valamilyen hosszúságú prefixe. Ez gyakorlatilag „korlátlan hosszúságú" előrenézést, vagy valamilyen formájú „szelektív" backtracking-et fog jelenteni, amivel a hagyományos $LL(k)$ elemzés felismerési erősségét növelhetjük, általános backtrack nélkül. Ezen technikával az alfejezet elején példának felhozott C++ elemzési kétértelműséget könnyedén kezelhetjük.

A szemantikus predikátum már nem csupán a szintaxisra hagyatkozik, hanem tetszőleges, az elemzést eldönthető információt használ fel. Ez tipikusan a bemeneten következő valahány szimbólumra vonatkozó szemantikus információ, mint például azok fajtája a szimbólum táblában. Az előző alfejezetben említettük, hogy egyes esetekben az elemzés iránya csak ilyen jellegű információk alapján dönthető el, amelyeket esetünkben a már felépült ASG-ből nyerünk. Erre egy példa a `(T)` szimbólumsorozat, amely a `T` azonosító típusának függvényében lehet típuskényszerítés vagy zárójelezett elsődleges kifejezés. Külön figyelmet igényel az a tény, hogy a szimbólumokra vonatkozó információ kinyerése az ASG-ből általában nem triviális, hiszen a nyelv különböző szabályainak megfelelő kereséseket kell elvégezni a gráfban, amelyek esetenként bonyolultak lehetnek.

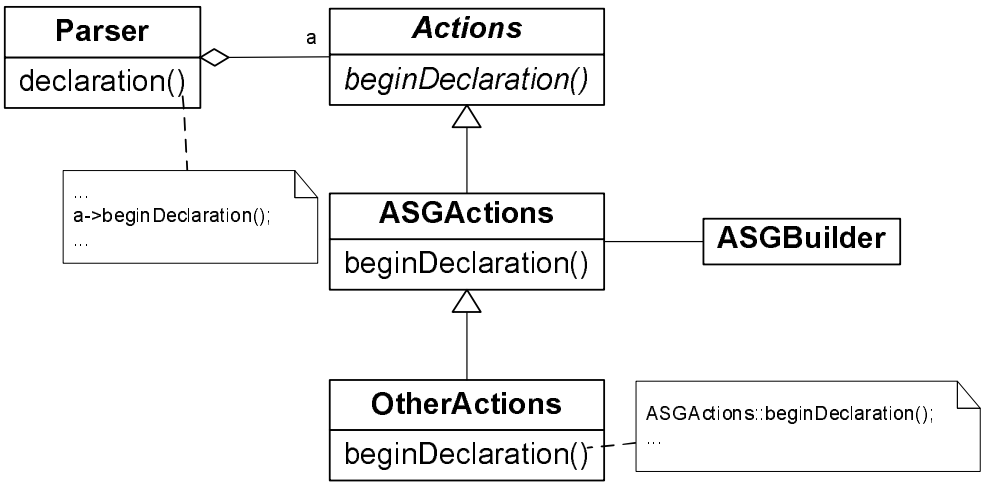
Az elemzés és a vele párhuzamos ASG építés normális menetét kicsit felborítja a C++ azon tulajdonsága, hogy egy osztályban definiált metódusban hivatkozni lehet a forrásban később definiált adattagokra, ami ellentétes azon alapelvvel, hogy minden használt nevet előbb deklarálni kell. Ennek áthidalására azt az elemzési trükköt használjuk, hogy egy osztály metódusainak részletes elemzését elhalasztjuk az osztály definíciójának végéig, amikor már minden adattag ismert. Ezt persze technikailag nem könnyű megoldani, hiszen a beolvasott tokeneket egy ideig ideiglenes pufferban kell tárolni.

Gyakorlatilag az elemző magját a nyelvtan a kisegítő rutinokkal és az ASG-t építő akciók képezik. Ezek kifejlesztése jelentette a legnagyobb munkát, amelynek az eredménye egy több mint 200 szabályból álló nyelvtan és egy meglehetősen összetett algoritmus halmaz, amely a sémának megfelelő forráskód-modell építését végzi. Ez utóbbival foglalkozik a következő alfejezet.

3.3. Forráskód-modell előállítás

Mint ahogy a fejezet elején azt már említettük, az ASG felépítése szintaxisvezérelt módon, az elemzéssel párhuzamosan történik. Az ilyen megoldás hatékonyabb, mint ha előbb ténylegesen fel is építenénk az elemzési (derivációs) fát, majd abból származtatnánk az absztrakt reprezentációt. Az alkalmazott rekurzív alászálló eljárásokat alkalmazó felülről lefelé haladó elemzés az építést logikus módon vezérelheti. Ehhez azt a mechanizmust használjuk, hogy a rekurzív eljárások végrehajtása közben minden lényeges ponton végrehajtunk egy akciót, amely implementációja végzi a tényleges építést. Akciókat tipikusan az alternatívák elején

és végén, valamint az egyes nemterminálisoknak megfelelő eljáráshívások körül helyezünk el. Az akcióhívások végrehajtása az objektumorientáltságot kihasználva egy általános felhasználást tesz lehetővé, amelyet az ASG építő is alkalmaz. A 3.3. ábrán illusztráljuk a mechanizmus végrehajtásában közreműködő részeket.



3.3. ábra. Az elemző szintaktikus akció-interfésze

A fenti osztálydiagramon a szintaktikus elemzést megvalósító **Parser** osztály egy rekurzív eljárását képviseli a *declaration*, amely egy adott pillanatban meghívja a megfelelő akciót, ami nem más, mint az absztrakt ***Actions*** osztály megfelelő absztrakt *beginDeclaration* metódusa. Ez normál esetben az **ASGActions** osztályban van megvalósítva, mégpedig az ASG építéséhez szükséges tevékenységgel. Ha egyéb dologra is szeretnénk használni a szintaktikus akciókat (és az elemzőt), akkor származtathatunk az **ASGActions**-ből és felüldefiniálhatjuk a használni kívánt akciókat saját tevékenységgel. Azonban nem kell elfelejteni első lépésként az ős-metódust meghívni, hogy az ASG építés párhuzamosan rendben folytatódjon. Ezt a mechanizmust használtuk fel a dinamikus szeletelő forráskód instrumentáló részében is (erről a dolgozat második részében esik szó).

A front end által előállított modell a Columbus sémának felel meg, amelyet két részben határoztunk meg. Az első az előfeldolgozóra vonatkozó információkat tárolja [101], a második a nyelv szintaxisát foglalja magában, ami tulajdonképpen a jelentősebb rész (ennek felel meg az ASG is). Az előfeldolgozó hasonló technológiát használ a modelljének felépítéséhez, mint amit a jelen alfejezetben ismertetünk, ezért annak részleteitől itt eltekintünk (az említett cikkben egy példa modellt is bemutatunk). A két modell együtt fogja képezni az analizálandó program teljes reprezentációját, ezeket egyes alkalmazásokban együtt kell használni. Jelenleg kutatómunkát folytatunk arra vonatkozóan, hogy definiáljunk a két séma összekötésére egy jól használható módszert, hogy a közös használatot megkönnyítsük.

A szintaktikus séma az elemzővel egyidejűleg fejlődve érte el a mai, stabilnak tekinthető formáját [29, 38]. A C++ séma kifejlesztését jelentős kutatás előzte meg, megvizsgáltuk a hasonló célú egyéb sémákat is (mint amilyen pl. a *Datrix* [7, 76] és a *cppML* [75]). A tervezéskor úgy döntöttünk, hogy a sémát a szabványos UML osztálydiagram jelölést felhasználva adjuk meg [79], aminek eredménye egy hat csomagból és több mint száz osztályból álló leírás, ahol minden osztály valamilyen nyelvi szintaktikus egységet reprezentál. A UML használata előnyösnek bizonyult a könnyebb érthetőség és az egyértelmű implementáció szempontjából. A séma megvalósítása a formális UML leírást követi, és programozási

interfészek (API) állnak rendelkezésre C++ és Jáva nyelven a modellek elérésére. A CAN program az építést a C++ API-n keresztül végzi, egy összefogó, ún. *factory* osztályt használva. A moduláris felépítésnek még az az előnye is megvan, hogy szükség esetén a séma egyes részeit specializálással bővíteni tudjuk, sőt egész csomagokat is lecserélhetünk saját megoldásúakra (ez például dialektus támogatásnál lehet hasznos). A [29] cikkben egy minta modellt is bemutatunk.

Az építés során számos segédstruktúrát és kiegészítő algoritmust kell használni, hiszen az elemző felől érkező akciók hívási sorrendje rögzített, ami nem mindig használható fel az azonnali építésre. Például egy deklaráció építésekor az összes deklarátornak ugyanazt a deklarációs specifikációt kell használnia, ezért az csak a deklaráció végén dobható el. Ezen kívül, a nyelv rekurzívan beágyazható jellege miatt számos helyen kell különböző vermeket használni az egyes elkészült elemek tárolására. Viszonylag összetett megoldást igényelt még a nevek feloldása, azaz a megfelelő entitás megtalálása egy névvel történő hivatkozás esetén. Függvényhívások esetén például ez magában foglalja az argumentumok típusainak meghatározását a standard konverziók figyelembevételével, a konstansok propagációját és végül a túlterhelt függvények feloldását. Ezen kívül számos egyéb kiegészítő feldolgozást is el kell végeznie az építőnek, amelyekre itt külön nem térünk ki.

A felépült ASG modellt az elemzés végén fájlba is kimentjük (különböző formátumokat is támogatunk, pl. tömör bináris és XML). A bináris fájl betöltésére a séma API nyújt támogatást, tehát a kapcsolódó analizátorok ezen keresztül érhetik el a front end által kigyűjtött információt. Ezután bármely magasabb szintű információ származtatása már elvégezhető a gráfból, hiszen az minden információt tartalmaz a forráskódról.

3.4. Speciális tulajdonságok

Ebben az alfejezetben áttekintjük, hogy milyen megoldásokat alkalmaztunk a front endünkben az előző fejezetben felvetett követelmények teljesítésére, amelyek egy általános célú analizátorra vonatkoznak. A teljesség és részletesség, valamint a kapcsolódás és front endhatár szempontjából már megvizsgáltuk a CAN elemzőt. Továbbá, az előfeldolgozás és felhasználói felület kérdésekkel is már foglalkoztunk.

A minél jobb *hibatűrés* elérése érdekében az elemzőbe beépítettünk egy, többnyire heurisztikákon alapuló mechanizmust, amely az esetleges elemzési szintaktikus hibák esetén lép életbe. A szintaktikus hiba lehet hiányos, hibás, vagy részleges forrás eredménye, például hiányzó fejlécfájlok miatt jelentkező ismeretlen szimbólumok, vagy nem befejezett és emiatt helytelen kód. Az általunk alkalmazott megoldásban hiba esetén nem lép ki egyszerűen az elemző, hanem a hibaüzenet után megpróbál helyrejönni és egy alkalmas következő pontnál folytatni az elemzést. Ezt a gyakorlatban két, párhuzamosan működő helyreállító algoritmus végzi. Az egyik a szintaktikus elemzőbe beépített speciális kivételkezelő, amely elemzési hiba észlelése esetén felfüggeszti az aktuális szabály feldolgozását és a bemenet továbbolvasását, és a rekurzívan hívott eljárások hívási vermét visszagörgeti egy olyan szabályig, amely valamely nagyobb nyelvi egységet valósít meg, ilyen például a deklaráció. Ugyanekkor, a bemeneti szimbólumok sorozatát szolgáltató pufferban is működésbe lép egy hibajavító mechanizmus, amely a szintaktikus elemzőtől kapott információ alapján megpróbálja az inputot is továbbléptetni egy megfelelő pontig, például a következő deklaráció kezdetéig. Ez gyakorlatilag a bemenet egy részének átugrását jelenti, nagy valószínűséggel

azét, amely a hibát okozta, így lehetőség nyílik a helyes folytatásra. A feldolgozás nélkül hagyott részek mennyiségének eldöntését különböző heurisztikákkal oldottuk meg. Ezek természetesen nem lehetnek minden esetben pontosak, viszont sok-sok kísérletezés eredményei, és általában nagyon hatékonyan működnek. Egy ilyen heurisztika az, hogy a következő deklaráció kezdetét úgy találjuk meg, hogy elmegyünk a következő pontosvesszőig, közben ügyelve a blokkzárójel-párok megfelelő kezelésére annak érdekében, hogy ne egy belső blokk belsejénél álljunk meg. Ha előbb jön egy záró blokkzárójel annak nyitó párja nélkül, akkor itt állítjuk meg a léptetést. Persze ez a módszer csődöt mond olyan esetekben, amikor a hiba éppenséggel a hiányzó zárójelek miatt van. Ilyen esetekre beépítettünk egy korlátot, ameddig a léptetést végezni kell, annak elkerülésére, hogy a teljes bemenetet átugorjuk.

Részleges forráskód elemzésének elősegítésére még egy további lehetőséget nyújtunk a felhasználónak. Ez abból áll, hogy lehetőség van megadni azonosítók listáját, amelyeket az elemző előre megadott módon fog kezelni, például típusnévként. Ez akkor lehet hasznos, amikor mondjuk egyes fejlécfájlok nem állnak rendelkezésre, és emiatt a benne lévő típusdefiníciók hiányoznak. Ugyanakkor, a felhasználó más forrásból tudhatja, hogy melyek azon nevek, amelyek típust jelentenek, sőt maga a konkrét típus ismerete nem is szükséges, a típus ténye elegendő. Mivel a szintaktikus elemzés egyik támpillére az azonosítók fajtájának ismerete, ez nagyban segíthet a hibák számának csökkentésében.

Az *analizálási sebesség* javítására két lehetőséget is kínálunk. Az első az, amikor a modell egyes részeire nincs szükség a további analízishez. Ekkor a teljes elemzés helyett egyes részeket átugrunk, és ezáltal jelentősen lecsökkentjük az elemzési időt. Nevezetesen, a függvények törzseinek elemzését kikapcsolhatjuk, amikor is az elemző csak a függvény fejlécét (szignatúráját) dolgozza fel, a törzset képviselő blokkzárójelek között lévő részt átugorja. Ekkor a modellbe nem fognak bekerülni az utasítások és kifejezések, ami egyes alkalmazásokban nem jelent hátrányt, mint például különböző architekturális információkat kigyűjtő analizátornál.

A másik lehetőség az analizálási idő csökkentésére az ún. *előfordított fejlécfájlok* (precompiled headers) használata. Ez a technika – amely a fordítóprogramok által is sűrűn használatos – főleg nagy rendszerek analízisénél előnyös. A módszer alapját az képezi, hogy számos esetben a forráskód implementációs fájljai nagy mennyiségű közös rendszerfájlt használnak, általában valamilyen gyűjtő fájlon keresztül, amely behozza a többi rendszerfájlt (például a Visual C++ rendszerek esetében ez az `StdAfx.h` fájl). Teljes rendszerek elemzését alapvetően úgy végezzük, hogy az analizátort (esetünkben a CAN-t) sorban behívjuk az összes implementációs forrásfájlra (előfeldolgozással). A rendszerfájlokban lévő összes kód analízise minden egyes fordítási egység esetén rendkívüli pazarlás, hiszen azok nagy része nem változik, és elegendő lenne a fejlécfájlokat egyszer leelemezni, majd a későbbiek során felhasználni a már előállított modellt. Ha ezt a megoldást választjuk, akkor a rendszer analízisekor egy kiválasztott fájlt használunk a teljes elemzésre, és ezt tudatjuk az elemzővel is megjelölve azon fejlécfájl nevét, amely a közös részt képviseli. Ekkor az elemző az előfeldolgozott bemenet közös részének elemzése és a modell előállítása után azt kimenti lemezre, és folytatja a maradék rész elemzését. Ezután az összes többi fordítási egység elemzésekor tudatjuk az elemzővel, hogy használnia kell az előfordított modellt. Ekkor az elemző a közös részt egyszerűen átugorja a fordítási egységben, és helyette betölti a korábban elkészített modellt. Végül, a megnevezett közös fejlécfájl után kezdi el a valódi elemzést úgy, hogy a közös részhez tartozó ASG már használható, mintha csak az is elemzés

eredménye lenne. Természetesen ekkor a lemezre kiírt modell már csak a közös rész utáni forrás tartalmát reprezentálja, tehát az így kapott modell fájlok a közös rész modelljével együtt lesznek csak teljesek. Megjegyezzük, hogy továbbra is lehetőség van egyes fordítási egységek teljes elemzésére, ha azok másmilyen rendszerfejléceket használnak.

A C++ elemző a standardnak megfelelő nyelvet ismeri fel, és a modellt is annak megfelelően építi fel. Az elemző ezen kívül támogatja még a Microsoft, a Borland és a GNU g++ fordítók *dialektusait*. Ezen dialektusok többnyire néhány új kulcsszóból, deklarációs módosítóból, esetleg új alap-típusból állnak. Ezen bővítéseket könnyen hozzá lehet adni a nyelvtanhoz, mégpedig úgy, hogy ezután mindegyik nyelvet egyidejűleg támogassa (szerencsére ezek között nem voltak egymásnak ellentmondó kiegészítések). A Borland és g++ fordítók esetében voltak komolyabb nyelvi átalakítást igénylő módosítások is, de ezek sem változtatják a „tiszta" alapnyelvet. Azonban a dialektus-támogatás a modellt nem érinti, az adott bővítések csak szintaktikailag lesznek felismerve, abból belső reprezentáció nem készül. Ekkor ugyanis a sémát is módosítani kellett volna, viszont megjegyezzük, hogy a séma modularitása miatt ezt viszonylag könnyen megtehetjük, ugyanakkor az eddigi alkalmazásokban erre nem volt szükség.

Analizálás szempontjából a C++ nyelv legnehezebb elemei közé tartoznak a *sablonok* (template). Egy tény-feltáró front end esetében mind maga a sablon, mind a sablonpéldányok modellezésére szükség van (a fordítóprogram a sablont eldobhatja miután az összes példányosítás megtörtént). A példányosításra különböző stratégiákat javasolnak, és a fordítóprogramok is különböző módszerek szerint dolgoznak. A nyelv meghatároz néhány kifinomult, sablonokkal kapcsolatos szerkezetet, amelyek ugyan hatékony, kompakt forráskódot eredményezhetnek, de feldolgozásuk rendkívül összetett (ilyenek például a kifejezés sablonparaméterek-támogatta meta-programozás, a parciális specializálás, függvény sablonok implicit példányosítása, osztály-tag sablonok, stb.). A fordítóprogramok is döntő többségükben csak részben támogatják az összes nyelvi lehetőséget, és valószínűleg igaz az a nézet, hogy az EDG front end az egyetlen a világon, amely 100%-os támogatást ad. Némelyik közismert és széles körben használt fordító is meglepően kis mértékben támogatja a sablonokat.

A mi front end-ünknek is ezen nyelvi elemekkel van a legtöbb nehézsége, az összes ismert hiányosság döntő többségében ezekhez kapcsolódik. Jelenlegi megvalósításunkban alap-üzemmódban nem végzünk példányosítást, csak a sablonokat és a specializálásokat tároljuk el a modellben. Azonban bekapcsolható egy példányosító üzemmód is, amely a sablonhasználatok nagy részét helyesen dolgozza fel, azonban egyes összetettebb eseteket nem kezel helyesen (pl. a standard sablon könyvtár némely részeit). Ezen kívül e módban az elemzés lényegesen költségesebb, ugyanakkor az alkalmazások többségében nincs is szükség ilyen jellegű információkra. A jelenlegi megoldásunk szerint a sablonokat a forráskód szintjén példányosítjuk, ellentétben az általános megoldással, amikor valamely szintű belső reprezentációt használják fel erre (általában a fát). A forráskód szintű példányosításnak megvan az az előnye, hogy az elkészült példányokat úgy látjuk, ami azoknak a valódi jelentése. Ha ez az üzemmód engedélyezett, akkor a valódi elemzés előtt végrehajtunk még egy megelőző menetet, amely az eredeti forráskódból kigyűjti a sablonokat és azok példányait, és közben minden egyéb, sablontól független kódot érintetlenül hagy. Ezután a példányokat legenerálja helyes forráskód szintaxissal, amit a második (valódi) menet tovább elemez. Ekkor az elemző már sablonok nélküli forrást fog látni (természetesen magukról a sablonokról is

készül modell egy kiegészítő elemzési fázisban). A közbülső, példányokat tartalmazó forrás a felhasználó számára is elérhető. További részletek a technológiáról a [36] cikkünkben találhatók. A jövőben tervezzük a példányosító megvalósításunkat tovább javítani, hiszen néhány valós alkalmazásban már problémát jelentett a hiányos kezelés.

3.5. Tapasztalati eredmények

A C++ front end számos alkalmazásban bizonyította alkalmasságát. Az analizálás sebessége, valamint a felhasznált memória kivételesen nagy rendszerek esetében is nagyon jónak mondható.

A [34] cikkben közzé tettünk három rendszerre vonatkozó mérési eredményt. A rendszerek az alábbi paraméterekkel rendelkeztek (nem-előfeldolgozott forrásra értendők):

3.1. táblázat. A teszt-rendszerek

rendszer	fájlok	bájtok	sorok
Jikes	77	3,5MB	94 611
Leda	508	2,9MB	116 752
Writer	9 449	66,5MB	1 764 574

A *Jikes* rendszer egy összetett osztályhierachiával rendelkező fordítóprogram, a *Leda* sablonokat használó gráf könyvtár, valamint a *Writer* a StarOffice irodai programcsomag egyik programja, közel tízezer forrásfájllal.

Az alábbi táblázatban láthatjuk a rendszerek teljes analíziséhez szükséges időt és memóriafogyasztást (800MHz-es Intel processzorral), valamint néhány kigyűjtött alapvető elem összes számát:

3.2. táblázat. Analizálási eredmények

rendszer	idő (ó:p:m)	memória	osztályok	névterek	operációk	attr.
Jikes	00:01:02	19MB	275	1	3 471	1 643
Leda	00:05:50	49MB	1 563	1	10 802	8 287
Writer	01:55:09	139MB	4 988	99	61 553	23 862

A fenti eredmények alapján elmondhatjuk, hogy a memória-foglalás közel lineáris a rendszer méretével (pl. az osztályok számát tekintve). Az elemzési idő már nem alakul ilyen kedvezően, de azt is vegyük észre, hogy a legnagyobb rendszer már valós méretűnek tekinthető, ennél nagyobb rendszerekkel nem igazán találkozhatunk a gyakorlati felhasználás során, ugyanis ennél nagyobb rendszereket úgyis alrendszerenként célszerű analizálni. Ugyanakkor az analizálási idő így is jó, a fordítási idővel összemérhető. A Columbus rendszerrel kapcsolatos legutóbbi kísérletben [37] a *Mozilla* rendszer lett elemezve teljes sikerrel. Ezen rendszernek hasonló méretei vannak mint a *Writer*-nek, és hasonló hatékonyság is lett elérve.

A [36] cikkben is ismertettünk néhány régebbi mérést, onnan érdemes kiemelni azon kísérletünket, amelyben az egyik viszonylag kis rendszert (8 fájlból álló) analizáltuk az elő-

fordított fejlécfájlok használatával és anélkül is. Az eredmény az lett, hogy a gyorsítás nélkül négy perc alatt történt meg az elemzés, vele meg egy percig tartott.

3.6. Összegzés

Ezen fejezetben áttekintettük a C/C++ front end-ünkben alkalmazott technológiákat, és azt, hogy milyen módon kezeltük az általános célú front end-eknek támasztott követelményeket. Láthattuk, hogy a C++ nyelv elemzési nehézségeit sikeresen megoldottuk, a forráskódból kigyűjtött információt jól definiált séma szerint állítjuk elő a további felhasználás érdekében, és még számos speciális szolgáltatást is beépítettünk a rendszerünkbe, segítendő a tetszőleges célú analizálási igényt. A következő fejezetben áttekintjük a front end alkalmazásait.

4. fejezet

C++ analizátor alkalmazásai

„Minden igazságot könnyű megérteni, miután felfedeztük. A lényeg hogy felfedezzük."

— Galileo Galilei

A szoftver-karbantartás, valamint a meglévő rendszerek modellezésének és újratervezésének elősegítésére a szakirodalom számos célra javasolja a forráskód-analizátorok használatát. Csupán említés szintjén, ide tartoznak: kód mérése metrikák számításával, vizualizálás, dokumentálás, programmegértés, tesztesetek vizsgálata, forráskód-transzformáció és -nyelvi átírás, stb.

Az előző fejezetben bemutatott C++ analizátor front end-ünk minden további analízis alapja, beleértve az egyszerűbb tény-feltárókat, amelyek például a kód formázott dokumentációját állítják elő, de a bonyolultabb számítást végző algoritmusokat is, mint amilyen az adatfolyam számítás. Mivel az előállított modell minden információt tartalmaz a forrásról, lehetséges lenne akár egy teljes fordítóprogram elkészítése is. Ugyanakkor, hogy egy teljes szoftverrendszert sikeresen analizáljunk, számos egyéb dologra is szükség van. Számunkra ezeket a Columbus keretrendszer nyújtja, amelybe olyan fontos dolgokat építettünk bele mint a projektkezelés, valamint a modellek összefésülése és szűrése. E rendszerről és használatairól beszélünk a következő alfejezetben, majd utána tárgyaljuk a front end egyéb, a keretrendszeren kívüli alkalmazásait.

4.1. Columbus keretrendszer

A Columbus keretrendszer [30, 31, 32, 33, 34] kifejlesztésének célja az volt, hogy egy olyan keretrendszert nyújtsunk, amely a fent említett célokat szolgálja, valamint moduláris felépítése és magasfokú bővíthetősége révén általános célra legyen alkalmazható. A keretrendszer összefogja és vezérli egy analizálási feladat minden tevékenységét, úgy mint a projekt felépítése, források egyenkénti analízise, a kapott modellek összefésülése és szűrése, és végül az adatok feldolgozása és transzformálása a kívánt célnak megfelelően. Ezzel együtt minden lényegi tevékenységet külső, a rendszerhez dinamikusan kapcsolódó eszközök végzik, így érve el a könnyű bővíthetőséget. Maguk az eszközök parancssori alkalmazások, és ezáltal a keretrendszertől függetlenül is használhatók, például az analizálandó rendszer fordítási folyamatába beépülve.

A Columbus nincs kötve adott programozási nyelvhez sem, jelenleg a C++-hoz tartozó front end és összefésülő is parancssori programként vannak behívva. Az analizátor a jelen

dolgozatban tárgyalt CAN program, amely a fentiek alapján a Columbus forrás-feldolgozási folyamatának elején lesz végrehajtva (az előfeldolgozás után) a projektben szereplő minden egyes forrásfájlra. Maga a keretrendszer nem ismeri az előállított modelleket, azokat nem dolgozza fel, csak tovább adja a következő fázisoknak.

A konkrét feldolgozási feladatokat is külső programok valósítják meg. A jelenlegi Columbus rendszer néhány hasznos feldolgozást tartalmaz a C++ nyelvhez. Az alábbi felsorolásban áttekintjük a sémának megfelelő információkat tartalmazó különböző támogatott formátumokat. Ezek mindegyike az elkészült modell megfelelő transzformálásával áll elő.

1. *PPML* és *CPPML*. Ezen formátumok az előfeldolgozó és C++ sémáknak megfelelő XML reprezentációk.

2. *GXL*. Ugyancsak XML-alapú, sémának megfelelő kimenet. A GXL formátum egy általánosan használt gráf-leíró, amelyet számos eszköz támogat [53].

3. *UML XMI*. UML szerkezeti (osztály) modell [79] XML-alapú fájlformátummal, a séma modell alapján. Az XMI dokumentumokat számos UML modellező eszköz be tudja olvasni, mint például a Rational Rose [85].

4. *FAMIX XMI*. A FAMIX modellnek megfelelő formátum, amelyet különböző eszközök használnak, például a *CodeCrawler* vizualizáló eszköz [25].

5. *RSF*. Az RSF fájlformátumot használja a *rigi* [78], és még néhány egyéb eszköz az adatcserére.

6. *VCG*. A VCG egy gráfokat leíró formátum, amelyet főleg gráf-vizualizálók használnak [90].

7. *HTML*. Böngészhető, forráskód dokumentáció, számos programmegértést támogató kényelmi szolgáltatással.

A fentieken kívül a Columbus rendszer képes előállítani még az alábbi, származtatott eredményeket is, amelyek a modellből számítódnak ki, és speciális célt szolgálnak.

1. *Metrikák*. Ez a feldolgozás közel 90 objektumorientált és hagyományos metrikát képes kiszámítani a modell alapján, amelyek az analizált rendszer különböző mérőszámait jelentik. Ezek többek között a minőség becslésére használhatók, mint például a [37] munkában.

2. *Tervezési minták*. A Columbus rendszer lett felhasználva a [6] munkában objektumorientált tervezési minták [42] felismerésére.

3. *Kód auditálás*. A Columbus-ban használt teljes analizálási technológia, beleértve a front end-et, az összefésülést és szűrést is, egy speciális eszközben is fel van használva, amely a forráskód auditálását végzi. A SourceAudit eszköz [31, 32] megvizsgálja, hogy a forrás eleget tesz-e bizonyos szabályoknak, amelyek lehetnek egyszerű formai előírások, de bonyolultabb szemantikai összefüggéseket tartalmazó kód-szerkezetek is. Az előbbi vonatkozhat például az azonosítók kinézetére vagy a programsorok tördelésére, míg utóbbi sokszor veszélyes szerkezetekre vonja fel a programozó figyelmét. A szabálysértések listáját a kódot a front end-del analizálva, majd a modellt feldolgozva

készítjük el, végül azokat a felhasználónak szolgáltatjuk. Az eszköz nem a keretrendszeren belül üzemel, hanem a fejlesztési eszközbe közvetlenül integrálódva, ami által a felhasználónak egy nagyon kényelmes használatot biztosít.

A Columbus rendszer és annak kimeneti eredményei számos egyetemi együttműködésünk tárgya, például a Maisa projekten belül tervezési minták felismerése [35], a FAMOOS projektben metrikákkal kapcsolatos vizsgálatok [88], vagy a GUPRO eszközzel való integrálás [27]. Akadémiai felhasználóból szerte a világon eddig több mint 600-at regisztráltunk.

4.2. További lehetséges alkalmazások

A Columbus rendszertől függetlenül is a front end számtalan alkalmazására nyílik lehetőség: minden olyan területen, ahol C++ elemzésre van szükség. Tervezzük egy kísérleti fordítóprogram kifejlesztését, amelyben különböző magas szintű optimalizálási algoritmusokkal szeretnénk kísérletezni. Ezen a téren van már némi tapasztalatunk a GCC fordítóval végzett munkánk révén [9, 11, 73] és a kódméret csökkentésének lehetőségeivel kapcsolatosan [10]. Itt természetesen számos feladat vár ránk, például az ASG-ből származtatni a szükséges program-reprezentációkat, például a folyamgráfot. A folyamgráfot ezután ki lehet egészíteni különböző függőségi gráfokkal, amelyeket például a statikus szeletelésnél tudunk majd alkalmazni. Ezen kívül érdekes kutatási terület még a forráskód nyelv-közti fordító (tolmácsoló), amely C++ nyelvű programból más nyelvű, de ekvivalens kódot generál, pl. Jávát.

Az előző fejezetben ismertetett szintaktikus akciók interfészét felhasználva az elemző további alkalmazására van lehetőség. Effajta bővítéssel eddig különböző *forráskód instrumentálási* alkalmazásoknál kísérleteztünk. Instrumentáláskor az elemzett forráskódba bizonyos pontokon szondákat építünk be, amelyek – a kiegészített kódot lefordítva – adott tevékenységet végeznek a program eredeti viselkedését is megtartva. Egy konkrét alkalmazás a dinamikus hívási gráf előállítás volt, amikor minden eljárásba történő belépéskor és annak elhagyásakor feljegyeztük a hívó és hívott eljárást. Számos más célból is instrumentálhatjuk a kódot, például szövegformázásra, ahol az eredeti forrást adott módon megformázzuk, esetleg jelölőkkel egészítjük ki.

Ugyancsak instrumentálással érjük el a dinamikus programszeletelés alkalmazásában a végrehajtási nyom előállítását, továbbá a program függőségeit tároló struktúrát a forráskód modellből nyerjük. A dinamikus szeletelés témájával foglalkozik az értekezés második része.

4.3. Összegzés

Ezen fejezettel zárjuk az értekezés első részét, amely a C++ forráskód analízisének és a modell előállításának kérdéseivel foglalkozott. Áttekintettük a front end és a kapcsolódó keretrendszer eddigi alkalmazásait, és érintettünk néhány további lehetséges területet. Láttuk, hogy a programmegértéshez és egyéb területeken is létfontosságú egy általános analizátor megléte. Forráskód analízisen alapul a programmegértést támogató egyik konkrét eszköz is, a *programszeletelés*, amely témát a következő fejezettel kezdjük meg. Látjuk majd, hogy a megvalósított szeletelő módszereink is támaszkodnak az itt ismertetett analizátorra.

II. rész

Dinamikus programszeletelés

5. fejezet

Programszeletelés

„Ha másoknál távolabbra látok, ez amiatt van, mert óriások vállán állok.”

— Sir Isaac Newton

Az értekezés második részét egy bevezetéssel kezdjük a programszeletelés témájába. A szeletelés és a programszeletek fogalmát számos módon definiálták már az irodalomban, de az alapvető ötlet mindig ugyanaz: próbáljuk meg úgy csökkenteni a problémánk méretét, hogy az analizálandó programnak csak a vizsgálatunk szempontjából lényeges részét kelljen figyelembe vennünk. E módszer számos alkalmazásban megkönnyíti a programmegértést, ugyanakkor a szeletek meghatározása komoly számítási apparátust igényel. Valószínűleg ez az egyik oka annak, hogy még mindig túlsúlyban vannak az elméleti módszerek, illetve a csak prototípus szintjén lévő megvalósítások. A dinamikus szeletelés területén végzett munkánk némileg kiegészíti a gyakorlatban is használható módszerek palettáját.

5.1. Fogalmak

A *programszeletelés* (program slicing) [97, 103] egyfajta programanalízis, amelyet számos szoftverfejlesztési és -karbantartási feladathoz használhatunk. A teljesség igénye nélkül, ide tartoznak a program verifikáció [4, 49, 89], karbantartás [41], újratervezés [109], programmegértés [14, 50] és nyomkövetés [2, 57]. Általánosan szólva, a *program szelet* a programnak egy olyan részhalmaza, amely tartalmazza azon utasításokat, melyek közvetlenül vagy közvetetten kihatással voltak vagy lehetnek egy adott program-pont adott változóelőfordulásainak értékeire (ezen előfordulások és program-pontok alkotják a *szeletelési kritériumot*). A részprogram meghatározásának folyamatát és annak módszerét nevezzük *szeletelésnek*.[1] Ez azt jelenti, hogy az áttételes függőségeket is figyelembe véve, a csökkentett program a nevezett változókat azonosan fogja kiértékelni az adott előfordulásnál. Sőt, egyes szeletelési algoritmusok ténylegesen végrehajtható programot állítanak elő (*végrehajtható szelet* – executable slice), míg mások tetszőleges részprogramot produkálhatnak. Ugyanakkor, a végrehajthatóságot csak néhány alkalmazás követeli meg, itt viszont számolni kell azzal a ténnyel, hogy ezen szeletek kevésbé precízek.

[1] Ez a szeletelés általános értelmezése, amit még *hátrafelé irányuló szeletelésnek* is nevezünk (backward slicing). Ezzel szemben, *előre irányuló szeleteléskor* (forward slicing) azon utasításokat határozzuk meg, amelyek egy adott program ponttól függnek. A jelen értekezésben a hátrafelé irányuló szeleteléssel foglalkozunk.

Ha a részprogramot úgy határozzuk meg, hogy az a program bármely futásakor fellépő viszonyokat magában foglalja, akkor *statikus szeletelésről* beszélünk. Ha viszont csak egy konkrét futáshoz tartozó viszonyokat kell tartalmaznia, akkor az *dinamikus szeletelés*. Következésképpen, minden dinamikus szelet az adott szeletelési kritériumhoz tartozó statikus szelet részhalmaza. A fentiek alapján, a *statikus szeletelési kritérium* egy program-pontot és az ott szereplő változók egy részhalmazát tartalmazza, míg a *dinamikus szeletelési kritérium* egy konkrét program-futás paramétereit is magában foglalja így egy tesztesetet képezve (egy adott program bemenet), valamint a kérdéses utasítás egy konkrét előfordulását a program futása során (mivel ugyanazon utasítás többször is szerepelhet a végrehajtás során, a különböző előfordulásokat utasítás helyett *akciónak* fogjuk nevezni, amely magában foglalja az utasítás előfordulási sorszámát is). A tesztesethez tartozó programfutáskor végrehajtott akciók sorozatát és egyéb végrehajtási információkat az ún. *végrehajtási nyom* (execution trace) rögzíti.

Legyen szó statikus vagy dinamikus szeletelésről, a fenti általános definíció végtelen sok szeletelési módszerre ad lehetőséget, maga a teljes program megtartása is egy szélsőséges eset. Ezzel szemben, ha a részprogram azokat és csakis azokat az utasításokat tartalmazza, amelyek befolyásolják a kritériumban szereplő változók értékeit (bármely további utasítás eltávolítása már más értéket eredményezne a kritériumnál), akkor beszélhetünk a *precíz (minimális) szeletről*. Természetesen a gyakorlati szempont az, hogy a programszelet minél kisebb (precízebb) legyen, hiszen így lesz informatívabb. Sajnos, minimális szelet kiszámításához nem konstruálható általános algoritmus, gyakorlatilag minden használható módszer a precíz szeletnek csak egy konzervatív közelítését adja, amelyben lehetnek fölösleges utasítások. Ha e két véglet között az összes lehetséges statikus és dinamikus szeletelési algoritmus osztályait tekintjük, a statikus szeletelési algoritmusok osztálya részhalmaza a dinamikus szeletelési algoritmusok osztályának, hiszen minden statikus szeletet kiszámító algoritmus egy speciális dinamikus szeletet kiszámító algoritmus is egyben (természetesen gyakorlati szempontból arra törekszünk, hogy olyan dinamikus szeletelési módszereket adjunk meg, amelyek minél kisebb szeleteket produkálnak).

A fent leírt precíz szeleteknek inkább elméleti jelentőségük van (ld. például [13, 48]), és mivel a szeletelést a program értelme felől (kiszámított értékek) közelítik meg, ezért az ilyen szemléletet *szemantikus szeletelésnek* fogjuk nevezni. Itt a gyakorlati megvalósíthatatlanság abban rejlik, hogy azon kérdés, miszerint két, szintaktikailag különböző utasítás vagy függvény azonos eredményt hoz-e, általában nem eldönthető. Emiatt egy konkrét algoritmus – annak érdekében hogy helyes maradjon, tehát hogy a szóba jöhető utasításokat mind megtartsa – konzervatív módon vélhetően fölösleges utasításokat is belevesz a szeletbe. Tulajdonképpen a gyakorlatban is megvalósítható módszerek a program szintaktikáján alapszanak, így ezen módszereket a *szintaktikus* jelzővel fogjuk illetni. Jelen dolgozatban is két konkrét, szintaktikus szeletelő algoritmussal foglalkozunk.

Az évek során számos szeletelési módszer került kidolgozásra, amelyek különböznek egymástól az alkalmazási területeikben, valamint abban hogy a precíz szeleteket milyen módon probálják minél jobban megközelíteni. Némelyik módszer szándékosan enged a pontosságból a hatékonyság javára. Az ilyen közelítő módszerek lényegesen hatékonyabbak lehetnek, mint a pontos algoritmusok, viszont esetenként túlságosan is konzervatív eredményt szolgáltatnak (ld. például Agrawal és Horgan csökkentett PDG-alapú módszerét [3]). A módszerek alapvetően különböznek aszerint is, hogy azokat milyen típusú programozási nyelvhez fej-

lesztették ki. A legtöbb módszer magas szintű procedurális nyelvekhez készül, de léteznek például logikai programokat szeletelő algoritmusok is [96]. Számos módszer megadható alacsony szintű nyelvekhez is, például gépi kódú programok szeletelésére [21, 61]. A programszeletelésnek nagy irodalma van. Különböző tanulmányok készültek, amelyek áttekintik a módszereket és alkalmazásokat, például [23, 52, 58, 97].

A gyakorlati módszerek többsége a program elemei (változók, utasítások, címek, predikátumok, stb.) között fellépő különböző *függőségek* (úgy mint adatfüggések és vezérlési függések) alapján számolja ki a szeleteket [54]. Konkrétan, statikus szeletelési módszerek kidolgozása intenzíven folyik Weiser eredeti cikkének [103] megjelenése óta egészen a mai napig. Sokszor mélyreható részletességgel vannak az algoritmusok megadva, ezáltal a statikus módszereket elérhetővé téve gyakorlatilag bárki számára. Például Horwitz *és mások* [54] munkája számos későbbi implementáció és finomítás alapja lett (e módszert először Ottenstein és Ottenstein ismertette [81], amelynek az alapja az ún. program-függőségi gráf – PDG).

Ugyanakkor viszonylag kevés publikáció jelent meg, amely a dinamikus szeletelést a gyakorlati oldaláról közelíti meg, és részletes algoritmust ad meg. Konkrét implementációt adott meg például Agrawal [1], Kamkar [57] és Venkatesh [100]. Utóbbi még számos mérési eredményt is szolgáltat. Az, hogy a gyakorlatban is használt algoritmusokkal nem igazán találkozhatunk, jelentheti azt is, hogy a közölt módszerek nem alkalmasak valós méretű programok kezelésére.

Jelen értekezésben dinamikus szeletelési algoritmusokkal foglalkozunk magas szintű procedurális nyelvekhez, és megadunk két olyan algoritmust (a részletekkel egyetemben C nyelvű programok szeletelésére), amelyek valós alkalmazásokhoz is használhatók, hiszen lényegesen hatékonyabbak a korábbi módszereknél. A két algoritmus hasonlóan analizálja a programot és függőségeit, de ugyanakkor ellentétes elven működnek, és ezáltal különbözőek az alkalmazási területeik is. A következő alfejezetben áttekintjük a meglévő dinamikus szeletelési módszereket.

5.2. Dinamikus szeletelési algoritmusok

Számos alkalmazás szempontjából a dinamikus szeletek [3, 67] használata előnyösebb, mint a statikusaké, hiszen az előbbiek lényegesen kisebbek és ezáltal informatívabbak is. Például, nyomkövetésnél a lehetséges okát keressük egy hibának, amelyet egy adott programfutáskor észleltünk adott program ponton azáltal, hogy egy változó értéke hibás. Nyilván minél kisebb a program azon része amit át kell vizsgálnunk, annál hatékonyabb lesz a hibakeresés. A dinamikus szelet pontosan ezt az információt szolgáltatja. Ez a fajta szeletelés számos egyéb alkalmazásban is használható, amelyek közül néhányat a 9. fejezetben említünk.

Ellentétben a statikus módszerekkel, a meglévő dinamikus szeletelési algoritmusok nagyon szerteágazóak (néhány szerző Korel és Laski [66, 67], Gopal [43], Agrawal *és mások* [1, 3], Kamkar *és mások* [60], stb.). Az algoritmusok egy része végrehajtható szeleteket állít elő, amelyek szükségesek némely alkalmazásokban, de ugyanakkor kevésbé pontosak. Az algoritmusok aszerint is feloszthatók, hogy milyen formában tárolják és kezelik a függőségi információkat, valamint, hogy miként dolgozzák fel a dinamikus információkat a program futásáról. Általában a függőségek követésével fokozatosan épül fel a program szelet, de olyan megközelítés is létezik, amelynél a teljes programot fokozatosan csökkentik amíg lehet. Zhang *és mások* [105, 107] aszerint csoportosítják a dinamikus szeletelési algoritmu-

sokat, hogy azok legnagyobb számítási költsége hol jelentkezik. Az egyik eset az, amikor a program futása során fellépő viszonyokat először teljes egészében feldolgozzák a végrehajtási nyom alapján, majd a tényleges szelet-meghatározás csak ezután következik (*teljes előfeldolgozás*). A másik lehetőség az, amikor minden egyes szeletelési igény alkalmával a végrehajtási nyom újra feldolgozásra kerül (*előfeldolgozás mentes*). Természetesen mindkét esetnek megvannak az előnyei és hátrányai egyaránt, ezért a gyakorlati megvalósításoknak célszerű egy kompromisszumos megoldást találni és valamilyen *korlátozott előfeldolgozást* alkalmazni. Ez jelentheti például a végrehajtási nyom feldolgozásakor kiszámított összesített információk eltárolását adott időközönként és azok újrafelhasználását egy későbbi szeletelési igény esetén. A jelen értekezés keretein belül bemutatott két dinamikus szeletelési algoritmus alap, optimalizálás nélküli változatainak egyike megfelel a teljes előfeldolgozásos, a másika viszont az előfeldolgozás nélküli kategóriáknak. Az említett szerzők munkáját részben a mi algoritmusunk motiválta.

Az egyik legjelentősebb dinamikus szeletelési algoritmust Agrawal és Horgan fejlesztették ki [3]. Ők felismerték, hogy egy utasítás különböző előfordulásait a végrehajtás során különböző utasítás-halmazok befolyásolhatják (azaz, akciók közötti függőségeket vizsgálnak és nem utasítások közöttieket). Ez a felismerés lesz az alapja számos későbbi módszernek, beleértve a miénket is. Agrawal és Horgan négy algoritmust mutat be, amelyek a dinamikus információkat különböző módon veszik figyelembe és ezáltal pontosságukban valamint hatékonyságukban is eltérnek. Az első két módszer mindössze a hagyományos függőség alapú statikus szeletelési algoritmusokat egészíti ki bizonyos dinamikus információkkal. A másik kettő viszont már akciók közötti függőségek vizsgálatán alapul. A függőségeket az ún. *dinamikus függőségi gráf* (DDG) segítségével reprezentálják. A gráf egy csomópontja egy akciónak felel meg, a közöttük lévő élek pedig az akciók közötti dinamikus adat- és vezérlési függőségeket képviselik. A DDG egy irányított körmentes gráf. A dinamikus függőségeket még *megvalósult függőségeknek* is nevezzük, és adatfüggés esetén a valós adatfolyamot reprezentálják, illetve vezérlési függés esetén a konkrét vezérlés átadásáért felelős vezérlési függést értjük alatta. A DDG alapján már könnyű a dinamikus szelet meghatározása: a dinamikus szeletelési kritériumban szereplő akcióból kiindulva az élek mentén haladva bejárjuk a gráfot, és a közben érintett akciók utasításai fogják képezni a dinamikus szeletet. Az eredeti módszer legnagyobb hátránya az, hogy az így kapott gráf rendkívül nagy lehet, hiszen annyi csomópontja lesz, ahány akció szerepel a végrehajtási nyomban, az élek számát pedig a dinamikus függőségek határozzák meg, aminek eredménye a gráf gyakorlatilag kezelhetetlen mérete általános esetben. Ennek kiküszöbölésére a szerzőpáros egy módosított módszert is közzétesz, amelyben a gráf méretét a különböző lehetséges dinamikus szeletek számával korlátozzák. Sajnos, az így kapott gráf is rendkívül nagy lehet bizonyos programok esetén (a [97] cikkben ismertettek egy egyszerű programot, amelynek $O(2^n)$ darab különböző dinamikus szelete van, ahol n a program utasításainak száma). A módszer interprocedurális változatát az [1] dolgozatban ismertették.

Ugyancsak dinamikus függőségi gráfon alapuló algoritmust adott meg a közelmúltban Zhang és Gupta [106], amely a teljes gráf tárolása helyett annak egy kompaktált változatával dolgozik, és így megközelítőleg egy nagyságrendnyi javulást érnek el a tárigény szempontjából. Zhang *és mások* egyéb lehetőségeket is bemutatnak a tárterület csökkentésére a [105, 107] cikkekben. További, függőségi gráfon alapuló interprocedurális módszert adnak meg Kamkar *és mások* [59, 60].

Korel és Laski voltak az elsők, akik dinamikus szeletelési algoritmust publikáltak. Ez a módszer végrehajtható szeleteket számolt ki [66]. Ugyanezen szerzők adatfolyam egyenleteken alapuló iteratív algoritmusa a [67] dolgozatban található. A [69] cikkben Korel és Yalamanchili egy előrehaladó számításon alapuló módszert mutat be, amelyet később nemstrukturált vezérlésátadások kezelésére is alkalmas módszerré egészítettek ki [63, 64]. Ezen módszerek alapja az ún. eltávolítható blokk. Az alapötlet az, hogy a program futtatásakor minden blokkból történő távozás esetén az algoritmus eldönti, hogy a kérdéses blokkot bele kell-e venni a szeletbe vagy sem.

Korel és társszerzői alapvetően végrehajtható szeletek számításával foglalkoztak. Mint tudjuk, az így kapott szeletek általában lényegesen nagyobbak, mint a pusztán függőségeket figyelembe vevő módszerek (Venkatesh mérései szerint az arány körülbelül 2–3-szoros [100]), ugyanakkor egyes alkalmazásokhoz elengedhetetlen a végrehajthatóság (mint például tesztelés elősegítéséhez és újrafelhasználható komponensek meghatározásához, ld. például a [22] cikket). A mi munkánkban nem követeltük meg e tulajdonságot, így lényegesen pontosabb szeleteket kapva, hiszen mi csak a kérdéses programrészek elszigetelésében vagyunk érdekeltek, amely elegendő a szoftverkarbantartási feladatoknál és a programmegértésnél. Korel függőségi relációkon alapuló dinamikus szeletelési módszert is bemutat, amely alapötlete hasonló a mi igényvezérelt algoritmusunkéhoz, viszont a függőségek kezelése nem általános és kevésbé hatékony (ld. [64] és [66] cikkeket).

Mint ahogy korábban már utaltunk rá, konkrét megvalósítással is rendelkező dinamikus szeletelési algoritmusból nagyon kevés lelhető fel, például a következő cikkekben: [1], [57] és [100]. Azonban ezen implementációk használhatóságát nem igazolták valós környezetekben. Utóbbi munkában Venkatesh különböző algoritmusokkal kísérletezett és néhány kísérleti eredményt is bemutat. Konkrétan, négyféle megvalósítással végez kísérleteket, amelyek mindegyike C nyelvű programok szeletelésére alkalmas. Agrawal és Horgan függőségeken alapuló módszere [3], valamint Korel és Laski végrehajtható szeleteket számító algoritmusa [66] is kipróbálásra kerül. Sajnos, a cikk egyáltalán nem tér ki a programok megvalósításának részleteire, így a C nyelv szeletelésére vonatkozó specialitásokat sem tárgyalja. Ugyanakkor, a cikk néhány érdekes eredményt mutat be a dinamikus szeletek méreteivel kapcsolatban.

5.3. Egyéb szeletelési módszerek

A fentieken kívül még számos hibrid, illetve statikus szeleteléssel kombinált módszer ismeretes, úgy mint [45] *hibrid* szeletelése, az ún. *feltételes* szeletelés [16, 51] vagy a *kvázi-statikus* szeletelés [99]. Jelentős számú elméleti munka is napvilágot látott a statikus és dinamikus szeletelés témakörében, mint például az *amorf* szeletelés [50], de ezek kevésbé jelentősek a jelen értekezés szempontjából. Hasonlóan, nem foglalkozunk speciális technológiákhoz kidolgozott módszerekkel sem, mint például az objektum orientált [18, 71], konkurrens és osztott rendszerek szeletelésével sem [26, 65, 70, 108, 110].

5.4. Összegzés

E fejezetben áttekintettük a szeletelés témakörét, és kitértünk arra, hogy milyen problémákkal küzdenek a gyakorlatban is használható szeletelési módszerek. Ebből kiindulva adjuk meg az általunk kidolgozott dinamikus szeletelési algoritmusokat a következő fejezetekben, amelyek számos tekintetben hatékonyabbak az eddig meglévő módszereknél.

6. fejezet

Hatékony dinamikus szeletelési módszereink

„Minden legyen annyira egyszerű, amennyire csak lehetséges, de semmivel sem egyszerűbb."

— Albert Einstein

Az általunk kidolgozott dinamikus szeletelési algoritmusok lényegesen különböznek a már meglévő módszerektől. Algoritmusaink hatékonyan implementálhatók és ezáltal a gyakorlatban is használhatók valós programok analíziséhez. Ehhez az is hozzájárul, hogy a számításokhoz szükséges tárolt adatok mennyiségének csökkentésére törekedtünk azáltal, hogy egyszerű adatszerkezeteket definiálunk. E fejezetben két algoritmust adunk meg, amelyek összehasonlítása a 6.6. alfejezetben található.

6.1. Áttekintés

Algoritmusaink alapját a program-elemek közötti függőségek számítása képezi. Mindkét módszernél az utasítás-előfordulások (akciók) között dinamikusan fellépő adat- és vezérlési (kontroll) függőségeket követjük. A statikus függőség-alapú módszerek esetében minden lehetséges függőséget figyelembe kell venni (lásd pl. Horwitz *és mások* munkáját [54]), és e célból meg kell konstruálni a program-függőségi gráfot (PDG). A PDG csomópontjai a program utasításai, élei pedig az utasítások között potenciálisan fellépő függőségeket reprezentálják, azaz függőség minden olyan esetben létrejön, amikor a függő utasítást valamely utasítás befolyásolja *valamely* program-futás esetén. Ezzel szemben, dinamikus esetben, amikor *egy konkrét* futást vizsgálunk, csak a ténylegesen megvalósult (dinamikus) függőségeket kell figyelembe venni. Itt viszont nem elegendő csak magát az utasítást tekinteni, hanem annak egy konkrét előfordulására (akcióra) fogalmazzuk meg a függőségeket. Ez vezérlési függőség esetén a legutóbb végrehajtott akciót jelenti azok közül, amelyek az adott függő utasításra (statikus értelemben véve) potenciálisan kihatnak. A megvalósult adatfüggéseket pedig a változók utolsó definiálási program-pontjuk alapján határozzuk meg. Ezek alapján nincs szükségünk teljes statikus vagy dinamikus függőségi gráfra. Ehelyett sokkal egyszerűbb statikus struktúra kiszámítására van szükség, amelyet később mindkét algoritmus fel fog használni a szelet kiszámítása során.

A két bemutatott algoritmus a végrehajtási nyom és a statikusan kiszámított információk alapján számítja ki a dinamikus szelete(ke)t. Működésük szerint abban különböznek

egymástól, hogy milyen irányban dolgozzák fel a nyomot. Az első algoritmus *előre haladó* feldolgozást követ kezdve az elsőnek végrehajtott utasítással, majd a függőségeket a végrehajtás szerinti sorrendben viszi tovább. Közben minden egyes lépésben az éppen definiált változóhoz tartozó, az adott lépésben érvényes dinamikus függőségi halmazt határozza meg. Az így kapott halmazokat felhalmozza és felhasználja valamely más definiált változóhoz tartozó függőségek számításánál. Másszóval, ez az algoritmus a futás végére *globálisan* meghatározza az összes lehetséges dinamikus szeletet az adott futásra, és ezért nem is kap bemenetként dinamikus szeletelési kritériumot. Ezért a továbbiakban „globális" algoritmusként hivatkozunk rá. Az algoritmust először a [46] cikkben publikáltuk, amit követett a C-re való kiegészítés a [12] publikációban.

Ezzel szemben a második algoritmus pont ellenkező irányban dolgozza fel a nyomot, kezdve a dinamikus szeletelési kritériumban szereplő akcióval. Ezután a dinamikus függőségeket a nyomon *hátrafelé haladva* követi végig a nyom eleje felé. A hátrafelé haladó számítás egyébként a megszokott módja a függőségek követésének, például a DDG-alapú módszereknél is. Mivel ez az algoritmus egy konkrét dinamikus szeletelési kritériumra határozza meg a szeletet, a továbbiakban „igényvezérelt" algoritmusnak hívjuk. Amint azt látni fogjuk, mindkét algoritmusnak megvannak a maga előnyei és hátrányai, és ugyancsak az alkalmazási területei. A következő alfejezetekben az algoritmusok elvi formális leírását adjuk meg egyszerű programokra (csak skaláris változók, strukturált vezérlésátadások és intraprocedurális), míg a rákövetkező fejezet az algoritmusok megvalósítási részleteivel foglalkozik valódi C nyelvű programok analíziséhez. Jelen fejezetben a számításokat a 6.1. **(a)** ábrán látható példán keresztül mutatjuk be.

```
    #include <stdio.h>
    int n, a, i, s;
    void main()
    {
 1.   scanf("%d", &n);
 2.   scanf("%d", &a);
 3.   i = 1;
 4.   s = 1;
 5.   if (a > 0)
 6.     s = 0;
 7.   while (i <= n) {
 8.     if (a > 0)
 9.       s += 2;
        else
10.       s *= 2;
11.     i++;
      }
12.   printf("%d", s);
    }
```

(a)

```
    #include <stdio.h>
    int n, a, i, s;
    void main()
    {
 1.   scanf("%d", &n);
 2.   scanf("%d", &a);
 3.   i = 1;
 4.   s = 1;
 5.   if (a > 0)
 6.     s = 0;
 7.   while (i <= n) {
 8.     if (a > 0)
 9.       s += 2;
        else
10.       s *= 2;
11.     i++;
      }
12.   printf("%d", s);
    }
```

(b)

6.1. ábra. Példaprogram és annak dinamikus szelete az $(\langle \mathtt{a} = 0, \mathtt{n} = 2\rangle, 12^{15}, \{\mathtt{s}\})$ kritériumhoz

6.2. Jelölések

Az algoritmusok bemutatása előtt áttekintünk néhány alapvető definíciót és bevezetjük a használatos jelöléseket (alapvetően a [67] cikkre hagyatkozva némi módosítással és kiegészítéssel).

A program azon járható útvonalát, amely egy adott program-bemenet hatására végre lett hajtva *végrehajtási történetnek* nevezzük, röviden EH (execution history). Gyakorlatilag a végrehajtási történet a végrehajtáskor létrejövő akciók sorozata, kezdve a legelőször végrehajtott utasítással. Az adott végrehajtás során végrehajtott lépések (utasítások) összes számát J-vel jelöljük. Az akciók jelölésére az i^j-t használjuk, ahol i az utasítás programbeli egyedi sorszáma, j pedig a végrehajtási lépés sorszáma. Az alábbi jelölések is hasznunkra lesznek a továbbiakban:

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

Továbbá, $EH(j) = i^j$ megadja a j-edik lépésben szereplő akciót, valamint az abban szereplő utasítás-sorszámot még a következőképpen is jelöljük: $EHI(j) = i(EH(j))$.

A továbbiakban *végrehajtási nyom* alatt egy kiegészített végrehajtási történetet értünk. A kiegészítés különböző információkat jelent a végrehajtással kapcsolatban, úgy mint a memória címek használata. A végrehajtási nyomra az algoritmusok valós C programokra történő megvalósításánál lesz szükség.

Végül, egy dinamikus szeletelési kritériumot a $C = (\mathbf{x}, i^j, V)$ hármassal definiálunk, ahol $\mathbf{x}$ a program bemenet, i^j az akció amelyhez tartozó dinamikus szeletet kell kiszámítani és V az i utasításban szereplő változók egy halmaza (a statikus kritérium csak az i utasításból és a V változóhalmazból áll). Ezek után Agrawal és Horgan a dinamikus szeletre vonatkozó definícióját tekintve [3], a dinamikus szelet nem más, mint a program utasításainak azon halmaza, amelyek ténylegesen kihatással voltak a V halmazban szereplő változók i^j akciónál lévő előfordulásaik értékeire az $\mathbf{x}$ bemenettel történő futtatás során. Megjegyezzük, hogy a V-t általában úgy értelmezzük, hogy az az i utasításban a számításokhoz *felhasznált* változók halmaza. Ugyanis, ha az utasításnál *definiált* változó is a halmazban van, akkor az összes használati változóra, ha pedig csakis az összes használati változót tartalmazza, akkor egyben a definiált változóra is implicite vonatkozik a kritérium. Egy példa dinamikus szelet látható a 6.1. (**b**) ábrán (mellesleg, a megfelelő statikus szelet az egész programot tartalmazza).

6.3. Statikus fázis

Mindkét dinamikus szeletelési algoritmusunknak csupán egy kevés statikus információra van szüksége a szeletelni való programról. A statikus analízis fázisban egy speciális programreprezentációt számítunk ki, amit csak egyszer kell elvégezni az adott programra, majd ezt a közös ábrázolást használhatja fel mindkét algoritmus. A statikus fázis másik feladata a program instrumentálása, de erről később lesz szó.

Az említett programreprezentációt *D/U programreprezentációnak* hívjuk, hiszen definiálás-használat (*def-use*) viszonyokat ábrázol. A D/U reprezentáció az adatfüggéseket olymódon rögzíti, hogy csak a változók definiálásának és használatának tényét veszi figyelembe utasí-

tásonként, és nem tárolja a változó előfordulások közötti konkrét kapcsolatokat (más szóval, csak a változók nevei az érdekesek és nem az előfordulások). Ez lényegesen megkönnyíti a struktúra meghatározását, hiszen nem kell végrehajtani (az olykor nehéz és költséges) adatfüggőségeket számító eljárásokat, a definiálás-használat viszonyok könnyen kinyerhetők a program szintaktikai felépítéséből. Továbbá, az utasítások közötti közvetlen vezérlési függőségek is megadhatók ebben az ábrázolásban. A program egy utasításának D/U reprezentációja a következő formájú:

$$i.\ d:\ U\ ,$$

illetve az egész struktúra feltüntetve a következőképpen:

$$DU = \langle (d_1, U_1), (d_2, U_2), \ldots, (d_I, U_I) \rangle\ ,$$

ahol i az utasítás sorszáma és I az utasítások összes száma a programban. Az i-edik utasításban értéket kapó (definiált) változó a d, továbbá U a d kiszámításához i-ben felhasznált változók halmaza (használati halmaz). A továbbiakban a következő jelöléseket is használni fogjuk: $DU(i) = (d_i, U_i)$, $d(i) = d_i$ és $U(i) = U_i$ az i-edik utasításhoz.

Az algoritmusok tárgyalásához szükség lesz még a v változó *utolsó definíciójának* fogalmára is, amit $LD(v, j)$-vel jelölünk. Ez a függvény azt az akciót adja vissza, amelyben a v változó utoljára definiálva lett a j lépés előtt a végrehajtási történetben:

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

valamint használni fogjuk még a tömörebb $LD(v) = j(LD(v, j))$ és $LS(v) = i(LD(v, j))$ jelöléseket az utolsó definiáló lépés és utasítás megadására, a futás során aktuálisan érvényes j lépés sorszám mellett. Nyilvánvalóan $LD(v)$-nek és $LS(v)$-nek a futás minden lépésében vannak aktuális értékei, amelyek az adott változó definiálásakor kapnak új értéket és változatlanok maradnak az adott változó következő definiálásáig.

Az általunk használt D/U reprezentáció egyik fő erénye az, hogy segítségével a vezérlési függőségeket is azonos módon rögzíthetjük, mint az adatfüggőségeket. Ezáltal a szeletelési algoritmusok is egyszerűbbek lehetnek, mivel a kétféle függőséget azonosan kezelhetik. A hagyományos módszereknél nem használtak ilyen általánosítást. A vezérlési függőségeket a következő egyszerű ötlettel kezeljük. Minden definiált d_i és használt $u_k \in U_i$ változónak ($i = 1, \ldots, I$) lehet egy speciális jelentése, amit *predikátum változónak* hívunk. A predikátum változók virtuális változók, melyek ilyen formában nem szerepelnek a programban. Az i utasításnál definiált változó egy újonnan generált predikátum változó lesz (p_i-vel jelölve), ha az i utasítás egy *predikátum utasítás*, vagyis egy feltételes vagy ismétléses utasítás feltételét adja. A predikátum kiértékelésétől függ a feltételes ill. ismétléses utasítás törzse ill. magja végrehajtásának ténye. Továbbá, minden U_i halmaz tetszőleges i esetén a „normális" változok mellet tartalmaz még pontosan egy predikátum változót is, amely azt a predikátumot jelöli, melytől az i utasítás közvetlenül függ vezérlés szerint (direkt kontroll függés). A nemstrukturált ugró utasításoktól mentes (strukturált) programok esetében minden utasításnak pontosan egy vezérlési függés szerinti őse van. Predikátumok által nem tartalmazott (eljárás szintű) utasítások esetében ez a képzeletbeli *entry* predikátum, míg a beágyazott utasítások az őket szintaktikailag közvetlenül tartalmazó predikátumtól fognak függni. A jelen fejezetben tárgyalt algoritmusok elvi bemutatásához feltesszük ezt a tulajdonságot, bár

valós nyelvek esetén bonyolultabbak a vezérlési függőségek. Ezek kezelését a későbbiekben tárgyaljuk.

A 6.1. (a) ábrán szereplő példánk D/U reprezentációja a 6.2. ábrán látható. Itt p_5, p_7 és p_8-cal jelöltük a megfelelő predikátum utasításokhoz tartozó predikátum változókat, míg p^E az *entry* predikátum. o_{12} egy további virtuális változó, egy ún. *kimeneti változó*, amely a használt változok értékét kapja a kimeneti (kiíró) utasításban, és amelyre csak az algoritmusok könnyebb érthetősége miatt van szükség.

$i.$	$d:$	U
1.	$n:$	$\{p^E\}$
2.	$a:$	$\{p^E\}$
3.	$i:$	$\{p^E\}$
4.	$s:$	$\{p^E\}$
5.	$p_5:$	$\{p^E, a\}$
6.	$s:$	$\{p_5\}$
7.	$p_7:$	$\{p^E, i, n\}$
8.	$p_8:$	$\{p_7, a\}$
9.	$s:$	$\{p_8, s\}$
10.	$s:$	$\{p_8, s\}$
11.	$i:$	$\{p_7, i\}$
12.	$o_{12}:$	$\{p^E, s\}$

6.2. ábra. A példaprogram D/U reprezentációja

Láthatjuk, hogy a D/U programreprezentáció méretét és a felhasználásához szükséges műveletigényt tekintve rendkívül hatékony eleme lesz az algoritmusoknak. A 8. fejezetben ismertetünk néhány konkrét mérést e struktúra méreteire vonatkozóan.

6.4. Globális algoritmus

Első dinamikus szeletelési algoritmusunk a végrehajtási történetet előre haladó irányban dolgozza fel és közben a fellépő dinamikus függőségek alapján kiszámítja a dinamikus szeleteket minden olyan változóhoz, amely közben definiálódik. Az EH természetes feldolgozása lehetővé tesz olyan megvalósítást létrehozását, amely a számításokat a program végrehajtásával párhuzamosan végzi. Köszönve a D/U reprezentációnak, amely lehetővé teszi a vezérlési- és adatfüggőségek azonos módon történő kezelését, maga az algoritmus elve rendkívül egyszerűvé válik.

6.4.1. Az algoritmus

Az algoritmus működése a következő (ld. a 6.3. ábrát): adott P programhoz és annak végrehajtásához egy adott $\mathbf{x}$ inputtal, sorban feldolgozzuk a közben rögzített EH végrehajtási történet minden egyes utasítását, kezdve az elsővel.[1] Miközben egy $i.\ d : U$ utasítást

[1]Nem kell az utoljára végrehajtott lépést elérni, a feldolgozást tetszőleges lépésig végezhetjük, ha csak az adott lépés előtt fellépő szeletekre vagyunk kíváncsiak.

dolgozunk fel, kiszámítjuk a $DynDep(d)$ halmazt, amely tartalmazni fogja mindazon utasításokat, melyek kihatással voltak d-re abban a pillanatban, amikor i-t hajtottuk végre az EH j-edik lépésében. Másszóval, meghatározzuk d-nek az adott lépésben érvényes dinamikus (adat és vezérlési) függőségeit. E halmaz elemeit az utasításban használt változók aktuálisan érvényes – és egy korábbi lépésben kiszámított– függőségeiből (áttételes függések hozzáadása) és ezen változók utolsó definiáló utasításából (közvetlen függés hozzáadása) állítjuk össze. Az előbbi a megfelelő $DynDep$ halmazokat jelenti, az utóbbi pedig az LS értékeket. Az utóbbiakat könnyű nyomon követni, hiszen amikor az i utasítást hajtjuk végre, akkor $LS(d(i)) = i$, és ez az érték változatlan lesz egészen addig, amíg a $d(i)$ változó egy későbbi lépésben újra definiálódik (amelyben esetleg i-től különböző utasítás lett végrehajtva). Ebből az következik, hogy a végrehajtási történet minden lépésénél az éppen definiált változóhoz tartozó LS értéket definiáljuk felül, továbbá ez az értékadás szigorúan a definiált változóhoz tartozó $DynDep$ halmaz kiszámítása *után* kell hogy bekövetkezzen, hiszen amikor e halmazt határozzuk meg, akkor a használt változók korábbi függőségeit kell figyelembe venni, beleértve azt az esetet is amikor a definiált változó egyben használt is. Ekkor ugyanis a használat úgyszintén a korábbi utolsó definiáló utasítást kell hogy kövesse. Egy p_i predikátum változó esetén az $LS(p_i)$ érték a predikátum utolsó kiértékelésének helye, amely természetszerűleg minden esetben i lesz. Értelemszerűen, $DynDep(p^E) = \emptyset$ és $LS(p^E) = 0$ (helyes programoknál, ahol csak definiált változót használnak, minden más $DynDep$ halmaz és LS érték előbb definiálva lesz, majd utána felhasználva).

program GlobalisAlgoritmus$(P, \mathbf{x})$
input: P : program
$\mathbf{x}$: program bemenet
output: dinamikus szeletek minden $(\mathbf{x}, i^j, V_i)$ kritériumhoz
$(j = 1 \ldots J,\ V_i = U(i))$
begin
  EH rögzítése
  $LS(p^E) := 0$
  $DynDep(p^E) := \emptyset$
  for $j = 1$ **to** J
    $i := EHI(j)$
    $DynDep(d(i)) := \bigcup_{u_k \in U(i)} (DynDep(u_k) \cup \{LS(u_k)\})$
    $LS(d(i)) := i$
    $DynDep(d(i))$ kimenetre, mint $(\mathbf{x}, i^j, V_i)$ kritérium dinamikus szelete
  endfor
end

6.3. ábra. Globális algoritmus

Az algoritmust megvizsgálva látjuk, hogy az i^j akció végrehajtása után a $DynDep(d(i))$ halmaz pontosan a $C = (\mathbf{x}, i^j, V_i)$ dinamikus szeletelési kritériumhoz tartozó dinamikus szelet lesz, ahol a V_i halmaz az i utasítás összes használati változóját tartalmazza. Vegyük észre, hogy az eredeti dinamikus kritérium a használt változók egy tetszőleges részhalmazára vonatkozhat, ugyanakkor az algoritmus globális lévén minden használt változót figyelembe

vesz. Egy olyan alkalmazásnál, ahol az eredeti kritérium szerinti szeletet kell meghatározni, az algoritmus egyszerű kiegészítésével ez elérhető.

Összegezve, a globális algoritmussal a menet közben kiszámított $DynDep$ halmazokat a memóriában tároljuk, és ezáltal a hozzájuk rendelt definiált változókhoz tartozó, a végrehajtás egy adott lépésénél érvényes dinamikus szeleteket is. A végrehajtás többi lépésére vonatkozó szeletek a lemezre kerülnek kimentésre.

Ekvivalencia a DDG-módszerrel Vezessük be a következő relációt két akció között. Azt mondjuk, hogy egy i^j akció *dinamikusan függ* az $i'^{j'}$ akciótól, akkor és csakis akkor, ha $j' < j$, továbbá az algoritmus j-edik iterációjában $i = EHI(j)$, és valamely $u_k \in U(i)$ változóra $LD(u_k, j) = j'$ (ekkor pedig $i' = EHI(j')$). Más szóval, a j-edik iteráció akciója függ minden olyan korábbi j'-beli akciótól, amelyben a j utasításának valamely használt változójához tartozó utolsó definíció található. Ezen reláció segítségével pontosan Agrawal dinamikus függőségi gráfja építhető fel, feltéve, hogy a definiálás/használat viszonyokat azonos módon határozzuk meg. Agrawal a dinamikus szeletet úgy határozza meg, hogy a kritérium akciójából kiindulva bejárja az irányított élek mentén a DDG-t úgy, hogy az összes elérhető akciót érintse, végül az érintett utasítások adják a dinamikus szeletet. Ezután már könnyen látható, hogy a globális algoritmusunkkal előállított dinamikus szeletek megegyeznek Agrawal szeleteivel, hiszen az általunk előállított $DynDep$ halmazokba a fenti dinamikus függőségeket gyűjtjük, ami megfelel a gráf bejárásának.

Az algoritmus működését a 6.1. (a) ábrán szereplő példánkra alkalmazva mutatjuk be. A program bemenete legyen $\langle \mathtt{a} = 0, \mathtt{n} = 2 \rangle$, ami a következő végrehajtási történetet eredményezi: $EH = \langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 \rangle$ (a $j = 1 \ldots 15$ lépések jelölését elhagytuk). Az algoritmus futtatása során a következő értékek számítódnak ki:

akció	d	U	$DynDep(d)$	$LS(d)$
1^1	n	$\{p^E\}$	$\emptyset$	1
2^2	a	$\{p^E\}$	$\emptyset$	2
3^3	i	$\{p^E\}$	$\emptyset$	3
4^4	s	$\{p^E\}$	$\emptyset$	4
5^5	p_5	$\{p^E, a\}$	{2}	5
7^6	p_7	$\{p^E, i, n\}$	{1,3}	7
8^7	p_8	$\{p_7, a\}$	{1,2,3,7}	8
10^8	s	$\{p_8, s\}$	{1,2,3,4,7,8}	10
11^9	i	$\{p_7, i\}$	{1,3,7}	11
7^{10}	p_7	$\{p^E, i, n\}$	{1,3,7,11}	7
8^{11}	p_8	$\{p_7, a\}$	{1,2,3,7,11}	8
10^{12}	s	$\{p_8, s\}$	{1,2,3,4,7,8,10,11}	10
11^{13}	i	$\{p_7, i\}$	{1,3,7,11}	11
7^{14}	p_7	$\{p^E, i, n\}$	{1,3,7,11}	7
12^{15}	o_{12}	$\{p^E, s\}$	{1,2,3,4,7,8,10,11}	12

6.4. ábra. A globális algoritmus példa futása

Látható, hogy számos dinamikus szelet számítódik ki a futás során különböző dinamikus szeletelési kritériumokhoz. Például, a 12^{15} utolsó akcióhoz és az ott használt s változóhoz tartozó szelet nem más, mint $DynDep(o_{12}) = \{1, 2, 3, 4, 7, 8, 10, 11\}$, amit vizuálisan is ábrázoltunk a 6.1. (b) ábrán.

6.4.2. Komplexitás

A globális algoritmus műveletigényét alapvetően két tényező határozza meg: a fő ciklus iterációinak száma, ami minden esetben a végrehajtási történet lépéseinek száma (J), valamint a megfelelő $DynDep$ halmazok kiszámításának műveletigénye minden iterációban. A többi művelet konstans időben elvégezhető, beleértve a D/U reprezentáció kezelését is. Ehhez természetesen még hozzáadódik a dinamikus szeletek lemezre történő kiíratása, amennyiben erre szükség van adott alkalmazásnál. Minden iterációban $2 \cdot |U(i)|$ darab halmaz unióját kell kiszámítani: az egyelemű LS halmazokét és a megfelelő $DynDep$ halmazokét minden $u_k \in U(i)$ használati változóhoz. A halmazműveletek viszonylag költséges műveletek, függetlenül a megvalósítás fajtájától. Kiegyensúlyozott fa alapú megvalósítás esetén például két halmaz uniójának kiszámításához szükséges adatszerkezet összefésülése $O(n_2 \cdot \log(n_1 + n_2))$ műveletigényű, ahol n_1 és n_2 az első és második halmaz elemeinek száma.

A globális algoritmus műveletigénye arányos az $UN_1 + \ldots + UN_J$ összeggel, ahol UN_j a j-edik lépés halmaz-unió számítás műveletigénye. A fentieket alapul véve, a tagok a következők lesznek:

$$UN_j = (|U(i)| - 1 + D^j(u_1) + \ldots + D^j(u_{ki})) \cdot \log_2(|U(i)| + D^j(u_1) + \ldots + D^j(u_{ki}))\,,$$

ahol $D^j(v)$-vel jelöljük a $|DynDep(v)|$ értéket a j-edik lépésben. A legrosszabb eset meghatározásához vegyük észre, hogy a D^j értékek legfeljebb a program különböző utasításainak száma, I. Valamint azt, hogy minden $U(i)$ halmaz mérete legfeljebb a programban szereplő összes különböző változó száma, beleértve a virtuális változókat is (ezt V-vel jelöljük). Ezek alapján a legrosszabb esethez tartozó komplexitás $O(J \cdot V \cdot I \cdot \log(V \cdot I))$.

Ez a képlet valóban súlyosnak tűnhet, azonban a valós programokat tekintve, az I és V értékek használata túlzott. Először is, a használati halmazok átlagos mérete sokkal kisebb, általában 10 alatti, sőt ezen értékekre nem jellemző, hogy függnének a program méretétől ezért e tényező konstansnak is tekinthető (ld. méréseinket a 8. fejezetben). A $DynDep$ halmazok mérete is lényegesen kisebb, mint a teljes program (hiszen ezek dinamikus szeleteket képviselnek, amelyekről tudjuk hogy sokkal kisebbek a teljes programnál és általában azzal arányosak, amit ugyancsak bemutatunk a méréseinkkel). Ezek alapján a műveletigényt átlagos esetre egyszerűsíthetjük $O(J \cdot DS^{AVG} \cdot \log(DS^{AVG}))$-re, ahol DS^{AVG} a dinamikus szeletek átlagos méretét jelöli.

Ami a tárigényt illeti, a globális algoritmusunknak két adatszerkezetet kell tárolnia (a D/U reprezentáció mellett). Először, az LS értékek számára egy vektort, amelynek legnagyobb mérete V, valamint V darab $DynDep$ halmazt. Legrosszabb esetben a tárigény tehát $V + V \cdot I$ darab utasítás sorszám tárolása. Azonban a második tag átlagos esetben ugyancsak sokkal kedvezőbb, ezért a komplexitás $O(V^{DEF} \cdot DS^{AVG})$, ahol V^{DEF} azon különböző változók száma amelyek ténylegesen definiálva lettek az adott végrehajtás során, ami lényegesen kevesebb, mint az összes lehetséges változók száma (C esetén ez egy kicsit másképp alakul). Természetesen a $DynDep$ halmazok lemezen is tárolhatók, de ez futásidő hátránnyal járhat, hiszen tetszőleges halmaz betöltésére lehet szükség az algoritmus futása során. A kiszámolt dinamikus szeletek száma nyilván sokkal nagyobb – a végrehajtási történet hosszától függ – hiszen a változók különböző előfordulásaira más és más $DynDep$ halmazok kerülnek kiíratásra.

6.5. Igényvezérelt algoritmus

Az előző alfejezetben bemutatott globális algoritmussal sok dinamikus szelet határozható meg. Jelen alfejezetben egy olyan alternatív megoldást adunk, amellyel egy konkrét szelet adható meg igény szerinti dinamikus szeletelési kritériumhoz. Ezen algoritmus ellenkező irányban dolgozza fel a végrehajtási történetet, nevezetesen a kritériumban szereplő akcióval kezdődően és monoton haladva az első akció felé. Egyébiránt, a statikus fázis azonos a globális algoritmuséval, tehát a D/U reprezentáció is közös. A végrehajtási történet is azonos lehet, azzal, hogy az igényvezérelt algoritmus másik formában tárolja azt. Korel függőségi relációkon alapuló igényvezérelt algoritmusának alapötlete hasonló a miénkhez, viszont a függőségek kezelése nem általános és kevésbé hatékony (ld. [64] és [66] cikkeket).

6.5.1. Az algoritmus

Az algoritmus működése a következő (ld. a 6.5. ábrát): adott P program és $C = (\mathbf{x}, i^j, V)$ dinamikus szeletelési kritérium esetén[2] megkezdjük a végrehajtási történet feldolgozását kezdve az i^j akcióval. Ekkor az összes olyan akciót gyűjteni kezdjük, amelyek elérhetők a (D/U reprezentációból megkapható) közvetlen és áttételes függőségek által i^j-ből. Az elért akciókhoz tartozó utasítás-sorszámok fogják képezni az eredmény szeletet. E folyamat közben a dinamikus függőségek egy képzeletbeli hálóját építjük fel, amely azon utasításokban fog végződni, amelyek már nem függnek további akcióktól. Hogy pontosabban lássuk, tegyük fel, hogy már felfedtük azt, hogy a k^l akció közvetlenül vagy áttételesen befolyásolja i^j-t. Ekkor meg kell vizsgálnunk minden olyan változó előfordulást, amelyektől függ a k utasítás (tehát minden $u \in U(k)$ változót). Azon korábbi akciók, amelyekben a kérdéses u változók utoljára definiálva lettek, tovább viszik a láncolatot. Az utolsó definíció azon akciót jelenti, amelyben az u változó lett definiálva és amelynek végrehajtási lépés sorszáma l előtt van, úgy, hogy az a legnagyobb ilyen sorszám legyen.

Mivel a fenti eljárás szerint a végrehajtási történet tetszőleges pozíciójának elérésére szükség lehet, azt egy alkalmas formában kell eltárolnunk. Ezt a formát *végrehajtási történet táblának*, röviden EHT-nak (execution history table) hívjuk. A tábla gyakorlatilag a végrehajtási történet összes elemét tárolja j-vel bezárólag egy adott rendezés szerint. Az átrendezés aszerint történik, hogy az adott akcióban mely változó a definiált (ezen információ a D/U reprezentációnak része). A tábla sorai tehát az összes különböző (valós és virtuális) változóval vannak címkézve, amelyek definiáltként előfordulnak a végrehajtási történetben j-vel bezárólag. Ezután a sorokban végrehajtási lépés szerinti növekvő sorrendben egyszerűen fel vannak sorolva a lineáris EH azon akciói, amelyekben a sorhoz tartozó változó van definiálva. Az EHT szerkezete itt látható:

$$\begin{array}{c|llll}
d_1 & {i_{11}}^{j_{11}} & {i_{12}}^{j_{12}} & \cdots & {i_{1m_1}}^{j_{1m_1}}, & j_{11} < j_{12} < \ldots < j_{1m_1} \\
\vdots & & & & & \\
d_N & {i_{N1}}^{j_{N1}} & {i_{N2}}^{j_{N2}} & \cdots & {i_{Nm_N}}^{j_{Nm_N}}, & j_{N1} < j_{N2} < \ldots < j_{Nm_N}
\end{array}$$

Továbbá, $\forall d_k, d_l \; d_k \neq d_l$, és $d(i_{k1}) = d(i_{k2}) = \ldots = d(i_{km_k}) = d_k \; (1 \leq k, l \leq N)$, ahol

[2] Az egyszerűség kedvéért feltesszük, hogy $V = U(i)$ minden esetben igaz. Ahhoz, hogy változók tetszőleges $V' \subseteq U(i)$ halmazát kezelni tudjuk, az algoritmus triviális módosítására van szükség a **for** cikluson belül.

N a különböző definiált változók száma. Az $EHT(d, j)$ jelölést a táblázat d változóhoz és j végrehajtási lépéshez tartozó bejegyzésére fogjuk használni, ha ilyen létezik. Másszóval: $EHT(d, j) = {i_{kl}}^{j_{kl}}$, ahol $d(i_{kl}) = d$ és $j_{kl} = j$ valamely l-re. Világosan látható, hogy a táblázat akcióinak sorrendbe rakása j_{kl} szerint a lineáris végrehajtási történetet eredményezi.

A fent említett műveletet, amelyben egy adott d változó utolsó definícióját keressük meg adott j végrehajtási lépés előtt, a továbbiakban $EHT^{<}$ függvénynek nevezzük. Formálisan, $EHT^{<}(d, j) = {i_{kl}}^{j_{kl}}$, ahol $d(i_{kl}) = d$ és $j_{kl} < j$ valamely l-re, továbbá $\nexists m$, amelyre $j_{kl} < j_{km} < j$. Az EHT tábla fent leírt szerkezete lehetővé teszi az $EHT^{<}(d, j)$ függvény hatékony megvalósítását, hiszen a sor konstans időben megtalálható (a változó sorszámával indexelhető), a keresett akció pedig logaritmikus időben elérhető az adott sor hosszának függvényében. (A C-re történő megvalósításnál kicsit más lesz a helyzet.)

Ezután már részletesen is leírhatjuk az igényvezérelt algorimusunk lépéseit (ld. 6.5. ábrát). (A leírásban egy S halmaz e elemmel való bővítéséhez az $S \leftarrow e$ jelölést használjuk.)

program IgenyvezereltAlgoritmus(P, C)
input: P : program
$C = (\mathbf{x}, i^j, V)$: dinamikus szeletelési kritérium
(feltesszük, hogy $V = U(i)$)
output: S : P dinamikus szelete C-re

begin
  Végrehajtási történet rögzítése EHT formában i^j-vel bezárólag
  $S := \emptyset$
  $worklist \leftarrow i^j$
  while $worklist \neq \emptyset$
    k^l := legnagyobb l-lel rendelkező elem kivétele $worklist$-ből
    $S \leftarrow k$
    for $\forall u \in U(k)$
      $worklist \leftarrow EHT^{<}(u, l)$
    endfor
  endwhile
  S kimenetre
end

6.5. ábra. Igényvezérelt algoritmus

Az algoritmus kiszámítja a P program dinamikus szeletét a $C = (\mathbf{x}, i^j, V)$ dinamikus szeletelési kritériumra. Az első lépésben létrehozzuk az EHT táblát a P program $\mathbf{x}$ inputtal történő futtatása nyomán.[3] A szeletben szereplő utasításokat az S halmazban gyűjtjük. A függőségek követésére egy munka-halmazt tartunk nyilván, amelyet $worklist$-tel jelölünk és amelyben azon akciókat fogjuk tárolni, melyek még további feldolgozásra várnak. Egy akció

[3]Eközben elegendő a végrehajtási történetet az i^j akcióig tárolni, ha csak erre az egy szeletre vagyunk kíváncsiak. A teljes végrehajtási történet tárolására szükség lehet, amikor egymás után több, tetszőleges kritérium szerint is futtatni kívánjuk a szeletet kiszámító algoritmust. Továbbá, egy optimalizált megvalósításnál az EHT tábla kimentésével és folyamatos bővítésével elérhetjük azt is, hogy egy későbbi időpontban későbbi kritériumhoz is kiszámítsuk a szeletet, anélkül hogy a táblát teljes egészében újra fel kellene építeni.

feldolgozása során azt eltávolítjuk a halmazból és helyette az akció közvetlen függőségeit rakjuk bele. Mivel a függőségeket a kritériumban megadott i^j akciótól kezdve követjük, a $worklist$ is ezzel inicializálódik. Az algoritmus törzse egy **while** ciklusból áll, amely sorra veszi a $worklist$-ben még benne lévő összes akciót, majd akkor terminál, amikor nincs több függőben lévő akció és kiürült a $worklist$.

A ciklus törzsében tehát egy k^l akció $worklist$-ből történő eltávolítása után a hozzá tartozó k utasítás sorszámmal bővítjük az S halmazt (kivéve magát i-t, de az olvashatóság kedvéért ezt nem tüntettük fel az algoritmusban). Ezután megvizsgálunk minden olyan u változót, amelytől a k utasítás dinamikusan függ az adott lépésben, tehát az $U(k)$ halmaz elemeinek utolsó definíciójához tartozó akciókkal (ahogy azt az $EHT^<$ függvény visszaadja) bővítjük a $worklist$-et. Ezen eljárásból a p^E predikátum változót kihagyjuk, de az egyszerűség kedvéért ezt nem tüntettük fel az algoritmusban.

Ekvivalencia a DDG-módszerrel Az algoritmus helyességének igazolásához ismét vezessünk be egy relációt két akció között. Azt mondjuk, hogy egy a akció *dinamikusan függ* az a' akciótól, akkor és csakis akkor, ha az algoritmus egyik iterációjában a-t vettük ki a $worklist$-ből és ugyanebben az iterációban a'-t megpróbáltuk beletenni (lehet hogy már benne volt). Felhasználva ezen relációt, az algoritmus végrehajtása során egy képzeletbeli gráfot építhetünk az érintett akciókkal úgy, hogy minden a' akcióba húzunk egy bemenő élet az összes olyan a akcióból, amelyek tőle függnek. Ez a gráf tulajdonképpen egy irányított aciklikus gráf lesz (amit a továbbiakban DAG-gal jelölünk), mivel az $EHT^<$ függvény mindig olyan akciót ad vissza, amely szigorúan az argumentumban megadott akció előtt van, vagyis egy akció nem függhet egy nála későbbitől. Ha az algoritmus futását ezen gráf egy irányított bejárásaként tekintjük, akkor elmondhatjuk, hogy az biztosan terminál véges lépés után. Agrawal DDG-n alapuló módszerében ugyancsak az akciók között fellépő dinamikus függőségek alapján építi fel a gráfot, azzal a különbséggel, hogy azt előre haladva végzi, a programvégrehajtás irányával megegyezően. Ugyancsak feltesszük, hogy a definiálás/használat viszonyokat azonos módon határozzuk meg, így könnyen belátható, hogy a mi DAG gráfunk pontosan azon részgráfja a DDG-nek, amely a szelet meghatározásakor a bejárással keletkezik. Tehát a kiszámított dinamikus szeletek is megegyeznek. A gráfokra láthatunk példát a 6.8. ábrán.

Az algoritmus egyik finomítása az, hogy tetszőleges akció helyett mindig a legnagyobb lépés-sorszámút távolítjuk el a $worklist$-ből. Ezáltal a sorrend, amelyben az eltávolított akciókat érintjük, a DAG egy topológikus rendezésének felel meg, vagyis az eltávolított akciók lépés-sorszáma monoton csökkenni fog. Ezt a tulajdonságot kihasználhatjuk az algoritmus implementálása során úgy, hogy minden $worklist$ csökkentéssel az eltávolított akciót (és minden esetleges nála későbbit is, amely bele sem került a $worklist$-be) az EHT-ből is kivesszük. Egy másik, könnyebben megvalósítható stratégia a szükségtelen akciók eltávolítására az, amikor egy EHT sor elérése esetén a visszaadott akció utáni összes akciót abban a sorban törlünk.[4] E módszer által javítható az $EHT^<$ függvény átlagos hatékonysága ahogy haladunk előre a feldolgozással.

Az igényvezérelt algoritmus által elvégzett számításokat ugyancsak a 6.1. **(a)** ábrán szereplő példánkra alkalmazva mutatjuk be. Az $\langle \mathtt{a} = 0, \mathtt{n} = 2 \rangle$ bemenetre a végrehajtási tör-

[4]Ez megtehető, mert lévén hogy az argumentum és a visszaadott akció lépés-sorszáma között nem helyezkedik el másik elem a kérdéses definiált változóhoz, minden további akció az adott sorban az argumentum akciója utáni kell hogy legyen, ezekről viszont tudjuk, hogy törölhetők.

ténet EHT formája a következő lesz:

d	$\cdots$	$i_k{}^{j_k}$	$\cdots$
p^E	0^0		
n	1^1		
a	2^2		
i	3^3	11^9	11^{13}
s	4^4	10^8	10^{12}
p_5	5^5		
p_7	7^6	7^{10}	7^{14}
p_8	8^7	8^{11}	
o_{12}	12^{15}		

6.6. ábra. A példaprogram EHT táblája

A következő táblában láthatjuk az algoritmus fő ciklusának egymás utáni iterációiban kiszámított értékeket az $(\langle \mathtt{a} = 0, \mathtt{n} = 2\rangle, 12^{15}, \{\mathtt{s}\})$ dinamikus szeletelési kritériumra (az utolsó két oszlop a *worklist*-tel végzett műveleteket jelzi).

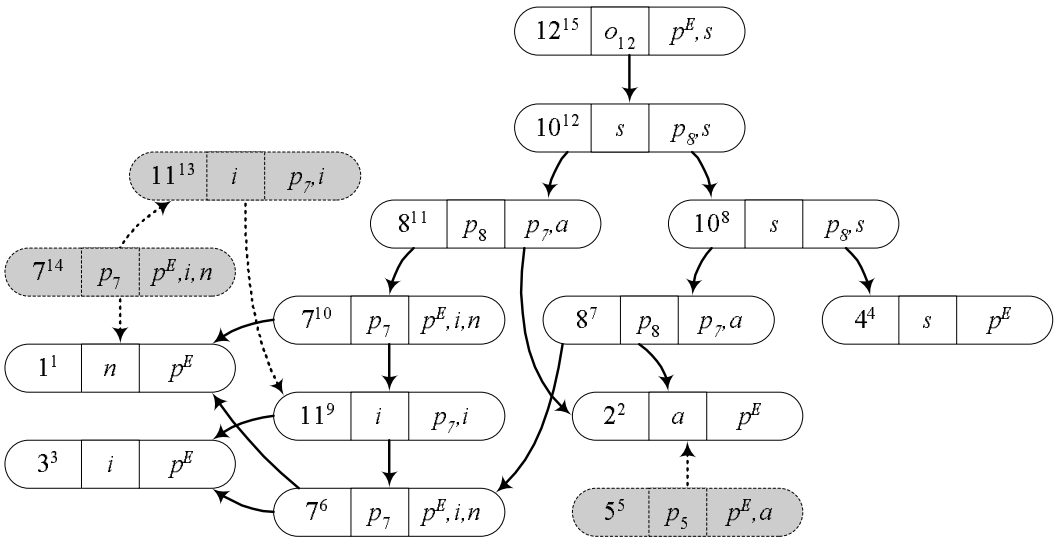
iteráció	S	*worklist*	eltávolítva	hozzáadva
0	$\emptyset$	$\{12^{15}\}$	—	12^{15}
1	$\emptyset$	$\{10^{12}\}$	12^{15}	10^{12}
2	$\{10\}$	$\{8^{11}, 10^8\}$	10^{12}	8^{11}, 10^8
3	$\{10, 8\}$	$\{10^8, 7^{10}, 2^2\}$	8^{11}	7^{10}, 2^2
4	$\{10, 8, 7\}$	$\{10^8, 2^2, 11^9, 1^1\}$	7^{10}	11^9, 1^1
5	$\{10, 8, 7, 11\}$	$\{10^8, 2^2, 1^1, 7^6, 3^3\}$	11^9	7^6, 3^3
6	$\{10, 8, 7, 11\}$	$\{2^2, 1^1, 7^6, 3^3, 8^7, 4^4\}$	10^8	8^7, 4^4
7	$\{10, 8, 7, 11\}$	$\{2^2, 1^1, 7^6, 3^3, 4^4\}$	8^7	7^6, 2^2
8	$\{10, 8, 7, 11\}$	$\{2^2, 1^1, 3^3, 4^4\}$	7^6	3^3, 1^1
9	$\{10, 8, 7, 11, 4\}$	$\{2^2, 1^1, 3^3\}$	4^4	—
10	$\{10, 8, 7, 11, 4, 3\}$	$\{2^2, 1^1\}$	3^3	—
11	$\{10, 8, 7, 11, 4, 3, 2\}$	$\{1^1\}$	2^2	—
12	$\{10, 8, 7, 11, 4, 3, 2, 1\}$	$\emptyset$	1^1	—

6.7. ábra. Az igényvezérelt algoritmus példa futása

Látható, hogy a futás végén az S halmaz a helyes eredményt tartalmazza (ld. 6.1. **(b)** ábrát), ugyanazt, mint amit a globális algoritmus is kiszámított. A fent említett DAG könnyen felépíthető az eltávolított és hozzáadott akciókkal, amely a 6.8. ábrán látható (az akciók mellett a definiált és használt változókat is feltüntettük). Továbbá, szaggatott vonallal még a teljes DDG hiányzó csomópontjait és éleit is jelöltük.

6.5.2. Komplexitás

Jelen alfejezetben a továbbiakban a végrehajtás hossza (J) a kritériumban szereplő akcióval bezárólag értendő. Az igényvezérelt algoritmus műveletigényének meghatározásakor az EHT tábla építésének költséget nem számoljuk, hiszen a tábla a végrehajtással párhuzamosan azonnal építhető és nem kell kétszer végigmenni a lépéseken, továbbá a tábla



6.8. ábra. Függőségi DAG a példaprogramhoz

elvileg lemezen is tárolható másik igény kielégítésére. A D/U programreprezentáció ebben az esetben sem jelent lényeges komplexitásbeli költséget.

Az igényvezérelt algoritmus műveletigényét alapvetően a következő tényezők határozzák meg: a fő ciklus iterációinak száma (IT), a legnagyobb elem eltávolításának ideje a $worklist$-ből (REM), az $EHT^{<}$ függvény végrehajtásának ideje ($EHTF$) és egy elem hozzáadásának ideje a $worklist$-hez (ADD). A műveletigény egy felső korlátja megadható a maximális értékekkel, azaz $IT \cdot (REM + \max(|U(i)|) \cdot (EHTF^{MAX} + ADD^{MAX}))$, ahol az $U(i)$ halmazok ($i = 1 \ldots I$) mérete legrosszabb esetben V, a különböző változók száma. Ezen tényezőket tovább vizsgálva, láthatjuk, hogy REM konstansnak tekinthető, hiszen a $worklist$-et célszerű rendezett adatszerkezetben tárolni. Továbbá, $EHTF$ logaritmikus a tábla adott sora hosszának függvényében (a sor kiválasztása konstans idejűvé tehető az elvi algoritmusnál), és ADD ugyancsak logaritmikus a $worklist$ aktuális méretéhez képest (feltéve, hogy rendezett struktúrát tartunk nyilván). Az utóbbi két tényező maximális értéke a végrehajtási történet hossza, de mint látni fogjuk, ez túlságosan erős felülbecslés. Az iterációk számáról annyit mondhatunk, hogy az legalább a kapott dinamikus szelet mérete kell hogy legyen, és legfeljebb a végrehajtott lépések számával megegyező lehet (mivel az iterációkban eltávolított akciók lépés-sorszáma szigorúan monoton csökken). A legrosszabb eset egy felső korlátja tehát $O(J \cdot V \cdot \log(J))$.

A másik megközelítésünk az, hogy a DAG éleinek számát használjuk fel az összes szükséges $EHTF$ és ADD műveletek számának meghatározásához. Definíció szerint, az algoritmus teljes futása alatt pontosan annyiszor kell ezen eljárásokat alkalmazni, ahány éle van a DAG-nak. Ez azt jelenti, hogy a fenti képlet két lineáris tényezőjét helyettesíthetjük $|E(DAG)|$-gal (ez viszont kevésbé jól becsülhető paraméter). Sőt, az is könnyen belátható, hogy ez utóbbi a kisebbik felső korlát, azaz $IT \cdot U^{AVG} = |E(DAG)| \leq J \cdot V$, hiszen $IT \leq J$ és $U^{AVG} \leq V$. Itt U^{AVG}-vel a menet közben felmerülő használt változók halmazainak átlagos méretét jelöltük. Továbbá, az is belátható, hogy IT-nek J mellett $|E(DAG)|$ is felső korlátja (minden iterációban pontosan egy elemet eltávolítunk a $worklist$-ből, ezért korábban összesen ugyanennyi elemet hozzá is kellett adnunk, tehát legalább ennyiszer próbálkoznunk is kellett a hozzáadással), amiből még az is következik, hogy $U^{AVG} \geq 1$. Sajnos IT két különböző felső korlátjának viszonyáról nem mondhatunk semmit, a gráf mérete

tetszőlegesen nagy lehet ($J \cdot V$ korlátig), de lehet lényegesen kisebb is J-nél, mint ahogy azt méréseink révén is bemutatjuk a 8. fejezetben. Tudjuk, hogy $|E(DAG)|$ a kisebbik felső korlát, de még ha $|E(DAG)| > J$ fenn is áll, a gráf akkor sem lehet V-szer nagyobb a lépésszámnál. Látni fogjuk, hogy átlagos esetben $IT < |E(DAG)| < J$ teljesül.

A legrosszabb esetnek megfelelő komplexitás logaritmikus tényezője két dolgot foglal magában, az $EHTF$-et és ADD-ot. Átlagos esetben az első műveletigényét jobban közelíthetjük a $\log_2(J/V)$ értékkel, feltéve a változók egyenletes eloszlását az EHT táblában. Sőt, a keresési idő folyamatosan javulni fog, ahogy haladunk a feldolgozással, ha alkalmazzuk az algoritmus leírásánál megadott optimalizálási technikát. ADD átlagos értékét valójában a $worklist$ mérete határozza meg, amiről tudjuk, hogy minden pillanatban $|worklist| \leq IT \leq J$. Fent láttuk, hogy IT kisebb értéket vesz fel, de a méréseink azt is igazolják, hogy $|worklist|$ átlagos értéke még kisebb. Igazából $|worklist|$ méreteit a dinamikus függőségek egymásra épülő struktúrája (DAG) határozza meg, és nem annak mérete. Egyes „sekély" gráfoknál a $|worklist|$ sok elemet fog tartalmazni, míg „mély" gráfoknál a méret kicsi lesz. Átlagos esetben azonban, amikor a gráf kiegyensúlyozottnak tekinthető, a $|worklist|$ mérete tükrözni fogja a gráfét. Sőt, méréseink azt is igazolják, hogy ezen érték inkább a dinamikus szeletek méretével van összefüggésben, amely általában még kisebb. Továbbá, ha eltekintünk az algoritmus azon finomításától, hogy a $|worklist|$-et rendezve tároljuk és mindig a legnagyobb elemet vesszük ki, a hozzáadás művelete konstans idejűvé is tehető.

A fentieket is figyelembe véve az átlagos eset műveletigényét $O(DS^{AVG} \cdot \log(J))$-ben határozhatjuk meg az alábbiak miatt. A lineáris V tényező itt is elhagyható, lévén hogy a használati halmazok átlagos mérete nem programméret-függő, hanem konstansnak tekinthető, valamint az iterációs számra vonatkozó méréseink alapján (ld. 8. fejezetet) az inkább az átlagos szeletmérettel van összefüggésben, mintsem a végrehajtás hosszával. A logaritmikus tényezőt sajnos nem csökkenthetjük tovább, annak ellenére, hogy ADD átlagos esetben kedvezőbb, az $EHTF$ a végrehajtás hosszának növekedésével tovább fog növekedni, a V osztó elveszíti hatását.

Az igényvezérelt algoritmus tárigénye az alábbiak szerint összegezhető. Az EHT tábla mérete azonos a lineáris végrehajtási történetével, kiegészítve a kereséshez szükséges struktúrák járulékos költségével. A D/U struktúra mellett további tárterületet foglal még a $|worklist|$, amely méretét a fentiekben tárgyaltuk, illetve az S halmaz, amely a kimeneti szeletet tárolja, ezért mérete nem jelentős. Összegezve tehát, a tárigény $O(J)$, az EHT tábla miatt, de ha azt nem számoljuk (lemezen is tárolható), akkor átlagos esetben $O(DS^{AVG})$.

Az igényvezérelt módszer hatékonyságának tárgyalásánál fontos az a tény is, hogy a végrehajtási történetet teljes egészében el kell tárolnunk, mielőtt a szelet kiszámításának nekilátnánk. Implementáláskor ezt a problémát úgy is áthidalhatjuk, hogy e struktúrát lemezen tároljuk, majd a betöltésnél az alábbiak szerint járunk el. Ha az algoritmus futása során szükségünk van egy j_{min} lépés-sorszámú akcióra, akkor blokkosítva betölthetjük a táblának csak a j_{min} feletti részét, ha az még nincs a memóriában. Ezen kívül, a $|worklist|$-ből eltávolított j_{max} akció feletti részeket törölhetjük, amint azt a korábbiakban tárgyaltuk. Bár a szükséges blokk mérete tetszőlegesen nagy lehet (j_{min} távolsága j_{max}-tól tetszőleges), átlagos esetben valószínűsíthető, hogy a függőségek lokalitása elég nagy, így a betöltött EHT részek kicsik és folyamatosan haladnak az első akció felé.

6.6. Összehasonlítás

Ezen alfejezetben összegezzük a két algoritmus algoritmikus és komplexitásbeli különbségeit, rávilágítunk mindkét módszer előnyeire és hátrányaira, valamint néhány szempontot is megadunk az alkalmazási területeikre vonatkozóan.

A legfontosabb különbséget az algoritmusok nevei is tükrözik, miszerint a globális algoritmus sok dinamikus szeletet állít elő (az összest), míg az igényvezérelt csak egy kritériumhoz tartozót. A másik szembetűnő különbség az, hogy míg a globális algoritmus a végrehajtási történetet előrehaladóan dolgozza fel, addig az igényvezérelt módszer a végénél kezdi a feldolgozást. Ez a különbség egy további nagyon fontos dolgot is jelent, nevezetesen azt, hogy a globális algoritmusnak nem kell eltárolnia a végrehajtási történetet, azt feldolgozhatja akár a végrehajtással párhuzamosan is. Ezzel szemben, a másik módszer esetében a történetet teljes egészében el kell tárolni mielőtt a tényleges szeletelést elvégezzük. Észrevehetjük azt is, hogy a globális algoritmusnak minden lépésben az összes fellépő dinamikus függőséget ki kell számolnia, az igényvezéreltnek pedig a kritériumban megadott akció szempontjából nézve csak az utolsó függőségeket (e különbség könnyen látható az algoritmusok futása során képzeletben felépíthető függőségi gráfokon is: a globális a teljes, Agrawal-féle DDG gráfot adná, míg az igényvezérelt csak a csökkentett DAG gráfot, ld. még a 6.8. ábrát). Továbbá, Zhang *és mások* szerinti felosztásban (ld. [105, 107] cikkeket) a globális algoritmus teljes előfeldolgozást végez, azaz előre kiszámolja az összes szeletet és lemezen eltárolja azokat, ami után egy konkrét szelet meghatározása triviális lesz. Ezzel szemben az igényvezérelt algoritmus előfeldolgozás mentes, tehát minden egyes igény kielégítésekor a függőségek számítását elölről kell kezdeni, bár a részeredmények újrafelhasználására létezik megoldás, amivel a jövőben foglalkozni is szeretnénk. Az említett szerzők munkájukban hivatkoznak a mi globális algoritmusunkra, és részben az is motiválta őket az igényvezérelt algoritmusuk kidolgozására (amely ugyancsak gráf-alapú), mellyel elkerülhető a teljes előfeldolgozás. Ezen eredményre viszont a mi igényvezérelt algoritmusunk a válasz.

Általános esetben nehéz összehasonlítani a két módszert a futási idő és tárigény szempontjából. Az első szembetűnő különbség az iterációk száma: a globális algoritmus pontosan annyi ciklusszámból áll, amilyen hosszú a végrehajtási történet, míg az igényvezérelt legfeljebb ennyit iterál. Sőt, méréseink alapján az utóbbi lépésszáma a szelet méretével van inkább meghatározva. Mindkét módszer magában foglal rendezett adatszerkezetekben való keresési műveleteket, de míg a globális algoritmus halmazműveletei a dinamikus szeletek méreteitől függnek, addig az igényvezéreltnél a keresendő struktúra méretét a végrehajtási történet hossza határozza meg. Az igényvezérelt algoritmus esetében a táblaépítésnek is költsége van, amit alapvetően a végrehajtás hossza határoz meg, de a tábla a végrehajtással párhuzamosan azonnal építhető és elvileg lemezen is tárolható másik igény kielégítésére. Ha nem nézzük a táblaépítés költségét, akkor világos, hogy egy szelet meghatározása ezen algoritmussal gyorsabb. Ha sok szeletet akarunk kiszámítani az igényvezérelt algoritmust egyenként lefuttatva, akkor az összköltség a szeletek számának egy határa fölött már nagyobb lesz a globális algoritmusénál. Igaz, ez a határ elég nagy lesz, közel az összes szeletek száma, ami a végrehajtás hosszával összemérhető. Természetesen, a minden egyes szeletelési igény esetén történő tábla-felépítés nem gazdaságos.

Ami a tárigényt illeti, a globális algoritmus által tárolt halmazok száma és mérete jelentős lehet. Az előállított szeletek száma a végrehajtási történet méretével nő, bár az összes

szeletet úgyis lemezen tároljuk. Amit memóriában kell tartani, az az összes definiált változóhoz az aktuális függőségi halmaza. A teljes tárigény tehát hosszú végrehajtások esetén is kezelhető, de még mindig nagyobb mint az igényvezérelt algoritmusé átlagos esetben, itt ugyanis egy futás során csak az egyetlen $worklist$-et kell tárolni, ami átlagos esetben nem jelentős, hiszen a szelet méretével arányos (ugyanakkor legrosszabb esetben a teljes végrehajtási történet méretét is jelentheti). Az utóbbi algoritmus további tárterület komplexitása az EHT tábla tárolása, de az némi futási hatékonyság-csökkenés árán megoldható lemezen történő tárolással is.

Összegezve, a globális algoritmust használjuk, amikor több szelet meghatározására van szükség egyszerre, például uniós vagy dekompozíciós szeletek számítására (ld. a 9. fejezetet). Amikor igény szerinti, egy dinamikus szeletelési kritériumhoz tartozó szelet meghatározása a feladat, mérlegelhetünk a két algoritmus használata között. Ilyen alkalmazás lehet például a tesztelő eszközökben, vagy nyomkövetőkben való felhasználás. Ha a végrehajtási történet teljes egészében történő tárolása nem célszerű, a globális algoritmust választhatjuk. Egyedi feladatokhoz általában az igényvezérelt algoritmus a jobb választás, kevesebb művelete és kisebb tárigénye miatt. Ugyanakkor, egyes hosszú végrehajtási történetekhez tartozó igény szerinti szelet meghatározására csak akkor alkalmazható hatékonyan ez a módszer, ha az EHT tábla tárolását és a benne történő keresést hatékonyan tudjuk megoldani. Ekkor, maga az algoritmus végrehajtása átlagos esetben kisebb költségű lesz, mint a globálisé, a kedvezőbb iterációs szám miatt. Sajnos, lehetnek olyan programok, amelyek esetében az igényvezérelt algoritmus iterációs száma is eléri a végrehajtási történet hosszát, ekkor megintcsak a globális algoritmus a hatékonyabb, az általában kedvezőbb logaritmikus komplexitási tényező miatt.

Az igényvezérelt algoritmus mentségére legyen szólva, hogy olyan alkalmazásoknál, amikor az algoritmust többször kell meghívni különböző kritériumokra és ezen kritériumok egymás után következnek a program végrehajtási történetében (például nyomkövetésnél), számos ponton optimalizálható a módszer. Például, a behívások között nem kell eldobni az EHT és $worklist$ struktúrákat, majd újra felépíteni azokat, ehelyett el is tárolhatók és folyamatosan módosíthatók. A már felfedett dinamikus függőségek is tárolhatók valamilyen ideiglenes tárban, és szükség szerint újra felhasználhatók, a költséges keresések helyett.

Ahhoz, hogy kísérleti úton megállapítsuk a két algoritmus alkalmazhatóságát nyomkövetési célokra, mindkettő megvalósítását tervezzük valamely nyomkövető eszközön belül.

6.7. Összegzés

A könnyebb érthetőség érdekében a dinamikus szeletelési algoritmusainkat két részletben ismertetjük. Jelen fejezetben megadtuk az alap-módszereket, amelyek egyszerű programok analízisére alkalmasak. Bármely valós programozási nyelv szeletelésekor szembe kell néznünk azonban számos problémával. A procedurális nyelvek esetén meg kell oldani az interprocedurális működést, a különböző adattípusok és vezérlési szerkezet kezelését. A következő fejezetben konkrét megoldást adunk a C nyelv sajátosságainak kezelésére, beleértve a mutatók és nem strukturált vezérlésátadások kezelését, hogy csak néhány problémát említsünk.

7. fejezet

Dinamikus szeletelési algoritmusok megvalósítása C nyelvre

„Könnyebb olyan nyelvben programozni, amelyben nincs minden benne, mint egyesekben melyekben igen."

— Dennis M. Ritchie

Az előző fejezetben bemutatott két, dinamikus szeletelésre alkalmas algoritmusunkat elvi szinten adtuk meg. A könnyebb érthetőség érdekében a módszerek lényegét egyszerű programokra adtuk meg, a vázolt algoritmusok csak skaláris változókra és csak strukturált vezérlésű, egy-eljárásos programok szeletelésére alkalmasak. E fejezetben megadunk az algoritmusok kiegészítésére vonatkozó minden olyan részletet, amelyekkel már valós C programok szeletelésére nyílik lehetőség.

Ami az algoritmusok gyakorlatban történő használatát illeti, alapvetően két különböző megközelítés lehetséges. Az első lehetőség szerint a program futásáról szóló szükséges (dinamikus) információkat – ami nem más, mint a végrehajtási nyom – egy ún. *instrumentált* program futtatásával érjük el. Itt az analizálandó program forráskódját[1] kiegészítjük olyan utasításokkal, melyek egyfajta „szondaként" képesek előállítani a szükséges információkat, de nem befolyásolják az eredeti program viselkedését. Az így módosított programot fordítás után futtatjuk a megfelelő bemenettel, és amellett hogy elvégzi az eredeti feladatát, előállítja a kívánt végrehajtási nyomot is. Az ilyen megvalósítás egy különálló szeletelő eszközként fogható fel, amely áll egy statikus analizáló modulból, egy instrumentálóból és a tényleges szeletelő modulból. Ezt a módot *offline* megvalósításnak fogjuk hívni.

A másik megvalósításról (*online* megvalósítás) akkor beszélünk, amikor a program végrehajtása önállóan történik valamilyen végrehajtási környezetből. Ez tipikusan a nyomkövető rendszer (debugger), amely a dinamikus információkat szolgáltatja. Itt nincs szükség instrumentálásra, a végrehajtási környezet rendelkezik minden szükséges információval. Természetesen, a statikus információk kiszámításához itt is szükség van egy statikus modulra. Ezután, a szeletelő modul lehet különálló program, vagy a futtatási környezet része is. Jelen dolgozatban az offline megvalósítás részleteit adjuk meg, a nyomkövetőben történő alkal-

[1] Megjegyezzük, hogy lefordított, tárgykódú programot is instrumentálhatunk, de mindkét lehetőségnek megvannak az előnyei és hátrányai egyaránt. A forráskód szintű instrumentálás a megoldás hordozhatóságát segíti elő, valamint a szeletelés eredményének visszavetítése az eredeti forráskódra is sokkal könnyebben megoldható. Ugyanakkor, a tárgykód instrumentálásával lényegesen kisebb romlással kell számolni ami a futásidőt illeti, és a könyvtári függvények kezelése is egyszerűbbé válik (ld. még [100]).

mazásról további információkat a 9. fejezetben adunk meg.

A globális algoritmus kiegészítését C nyelvű programok szeletelésére a [12] publikációban adtuk meg. Jelen fejezetben részletesebben tárgyaljuk a megoldásokat, néhol továbbfejlesztve azokat.

7.1. C nyelvvel kapcsolatos problémák áttekintése

Mindkét megvalósítás esetében számos sajátos problémával kell számolnunk az elvi algoritmusok kiegészítésekor valós C programok kezelésére. Az alábbiakban bemutatjuk, hogyan oldottuk meg a mutatók, tömbök, struktúrák, az eljárás-közti függőségek, a nemstrukturált vezérlésátadás, több fordítási egység, és egyéb problémák kezelését.

1. Ahelyett, hogy a skaláris változókat, mutatókat és más összetett objektumokat különbözőképpen kezelnénk, minden számítást *memória cellákon* végzünk. E megközelítéssel rendkívül leegyszerűsödik az említett programelemek kezelése, hiszen – ellentétben a statikus szeleteléssel, ahol minden eshetőséget vizsgálni kell – itt minden dinamikus információ rendelkezésre áll az aktuális programfutás objektumainak tárolásáról. Először mindent áttranszformálunk memóriacímekké: a skalárisoknak az aktuálisan felvett címeit, valamint a mutatók, tömb- és struktúra-elemek konkrét címértékeit használjuk, továbbá a szeletelési kritériumban szereplő változókat is címmé alakítjuk (megjegyezzük, hogy egy mutató indirekciós hivatkozás függőségei magában fogják foglalni magának a mutatónak és a hivatkozott memória cella függőségeit is).

   Ezután az algoritmusok ezen címekkel dolgoznak, mint „változókkal". Ennek a megközelítésnek az a hátránya, hogy szemben az elvi algoritmusainkkal, C esetében a számításokhoz felhasznált különböző változók (használt memória címek) száma (és címe, mint egyed név) csak dinamikusan lesz ismert. Emiatt a megvalósítás egyes helyein költségesebb adatszerkezeteket kell alkalmazni.

   Megjegyezzük, hogy a teljesen pontos megvalósításhoz a változók kezdőcímei mellett egyes ritka esetekben szükség lehet az adott változó méretének megfelelő *címtartomány* figyelembe vételére is, de ez jelentősen bonyolítja az algoritmusokat. Ilyen eset a típuskényszerítés olyan (kissé szabálytalan) használata, amikor az eredeti és típuskényszerített adatok átlapolódhatnak (ld. a 7.5. alfejezetet).

2. Mivel minden C utasítás (mellékhatással rendelkező kifejezés) egynél több objektumot is definiálhat, minden utasításhoz egy *definiálás-használat listát* rendelünk és nem csak egy D/U elemet, mint ahogy azt az elvi algoritmusunknál tettük.

3. A C struktúra típusú adatszerkezetekkel (*struct*) kapcsolatos olyan műveleteknél, ahol maga a struktúra szerepel (struktúra másolás) a kifejezést lebontjuk a struktúra mezőkre vonatkoztatott egyenkénti műveletekre, ezáltal biztosítva a mezőkre bontott részletezettséget. Ez pontosabb megoldás, mintha az egész struktúrát egyben kezelnénk. A beágyazott struktúrák esetében természetesen minden al-mezőt külön kell kezelni. Az *union* esetében minden mező-elérést az egész union eléréseként értelmezzük, így a függőségeket is eszerint határozzuk meg.

4. Az eljárásokon átnyúló (interprocedurális) függőségek kezelése viszonylag könnyen megoldható a memória cellás megközelítéssel, hiszen minden cella globális változóként tekinthető. A végrehajtási történet tartalmazni fogja az összes megvalósult eljáráshívást a fellépő sorrendiséggel együtt. Az aktuális hívási argumentumok és visszatérési érték speciális lokális változókként könnyedén kezelhetők.

5. A lokális változókat az aktuális hívási veremben szereplő címeik által kezeljük. Látni fogjuk, hogy a lokális blokk-láthatósági tartományokat (scope) dinamikusan is nyilvántartjuk, mert erre szükség lesz a használt lokális változók keresésénél. A globális változók kezelése egyszerű, hiszen címeik rögzítettek lesznek a program teljes futása során.

6. A nemstrukturált vezérlés-átadásokat (a *goto* és egyéb ugró utasítás) úgy kezeljük, hogy minden (statikus értelemben véve) lehetséges vezérlési függőséget felveszünk a D/U programreprezentációban. (Az elvi leírásunkban használt blokk-szerkezetű programok esetében a vezérlési függőségek a szintaxis alapján könnyedén meghatározhatók.) Ilymódon az utasítások egynél több predikátumtól is függhetnek, ezek kiegészített kezelését a későbbiekben tárgyaljuk.

7. A C-beli deklarációs programsorok annyiban érintik a szeletelési algoritmusokat, hogy valahányszor egy változó (vagy struktúra-mező) definiálása belekerül a szeletbe, a hozzá tartozó deklaráció programsorával is bővítjük a szeletet. Ezen kívül, a deklarációkhoz esetlegesen hozzátartozó kezdetiérték hozzárendeléseket (inicializálás) is hozzáadjuk a D/U programreprezentációhoz.

8. Az algoritmusok elvi leírásánál a végrehajtási történet fogalmát használtuk a végrehajtott utasítások rögzítésére. C programok szeleteléséhez további információkra is szükségünk van, úgy mint a változók címei, mutatók értékei, eljáráshívások, blokktartományok határai, stb. Ezen információkat a végrehajtási nyom rögzíti.

9. Minden valós program könyvtári függvényeket is használ, ezek pedig befolyásolhatják az interprocedurális függőségeket a kimeneti paramétereik és visszatérési értékeik, valamint mellékhatásaik révén. Ahhoz, hogy a szeletelés helyes legyen, a könyvtári függvények függőségeit is figyelembe kell venni, de mivel ezen függvények forráskódja nem mindig áll rendelkezésre, a függvények szabványos szemantikáját használjuk fel. Előre meghatároztuk az összes könyvtári függvény D/U programreprezentációját azok specifikációja alapján, majd minden feléjük irányuló híváskor a szeletelési algoritmusok ezen reprezentációkat is figyelembe veszik.

10. A valódi programok több forrásfájlból épülnek fel, jellemzően fejlécfájlokból (`.h`) és implementációs fájlokból (`.c`). A fordítóprogram utóbbiakból először előfeldolgozott fordítási egységeket készít felhasználva a fejlécfájlokat, majd tárgykódot, amelyeket végül összeszerkeszt a végső futtatható programmá. Hasonlóan, szeletelő algoritmusaink az előfeldolgozott forrásokon dolgoznak, az alkalmazott offline megvalósításban ezek kerülnek instrumentálásra és a D/U programreprezentációk is ezen vannak megadva. A programhoz tartozó összes fordítási egységben szereplő utasítások egyedileg lesznek sorszámozva egy globális számláló segítségével. Végül, szeleteléskor az összes

fordítási egységhez tartozó információ szerepet játszik. Ezáltal a több forrásfájlos programok egy egészként lesznek kezelve.

11. A szeletelési kritériumot és az eredményt is az eredeti forrásfájl soraira való hivatkozással célszerű megadni. Ugyanakkor, a memóriacímekkel történő számolás, az utasításonként esetlegesen több definiált objektum, valamint előfeldolgozott források használata miatt megfelelő leképezéseket kell használnunk.

7.2. Offline megvalósítás vázlata

Az általunk kidolgozott két dinamikus szeletelési algoritmus offline megvalósítása C programok szeletelésére a 7.1. ábra szerinti elrendezést követi.

A szeletelési módszerek négy fő fázisból állnak, amelyből az első három, a statikus analízis és instrumentálás, valamint az instrumentált program fordítása és futtatása közös. A statikus analízis fázisában az előfeldolgozott forráskódból kiindulva előállítjuk a program D/U reprezentációját eltárolva azt a lemezen, valamint instrumentáljuk a kódot. Ezután a szokott módon lefordítjuk az így kapott forrást, ezáltal előállítva az eredeti programmal szemantikailag ekvivalens futtatható programot, amelyet a kívánt teszteset szerint futtatunk (a teszteset a globális algoritmus bemeneteként, illetve az igényvezérelt algoritmus dinamikus szeletelési kritériumában megadott program inputtal azonosítható). Másik teszteset alkalmazásához a statikus és fordítási fázisokat nem kell megismételni.

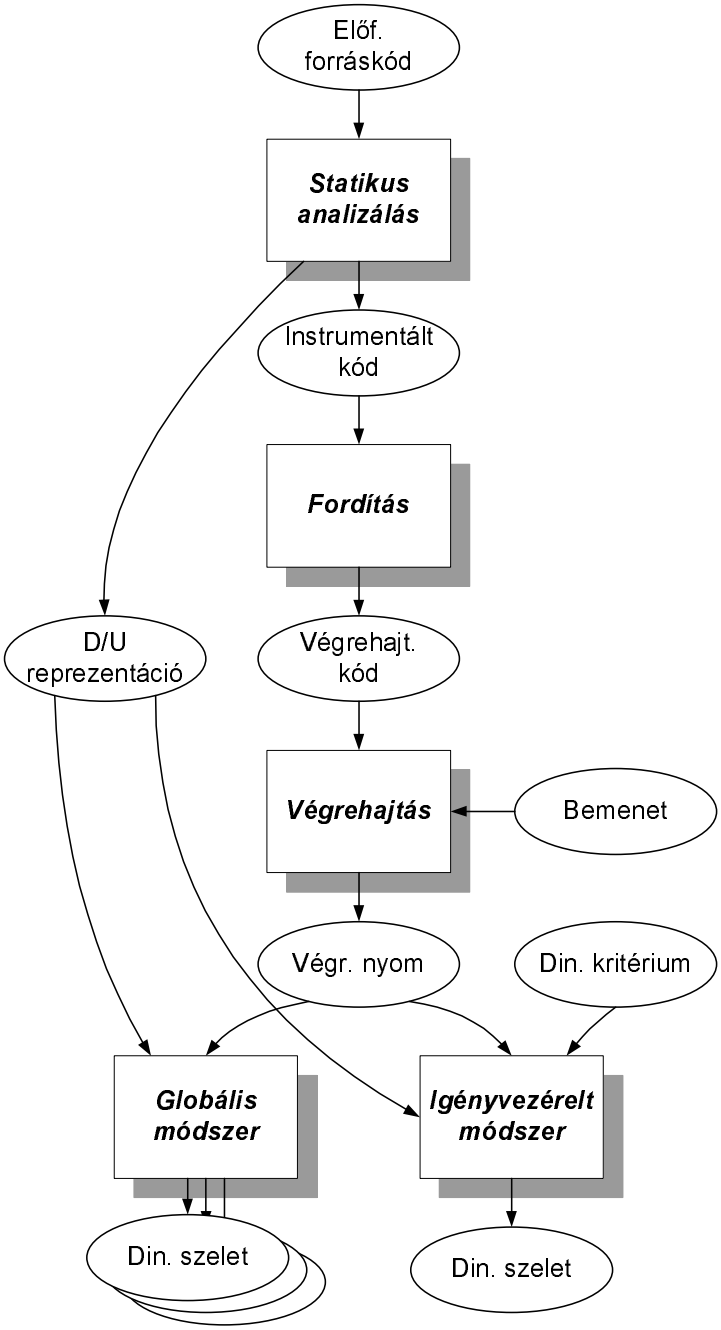
A futtatás során az instrumentálási kiegészítő utasításoknak köszönve előáll a végrehajtási nyom. A D/U programreprezentáció és a nyom felhasználásával már végrehajtható a globális és az igényvezérelt szeletelési algoritmus. Az előbbi nem igényel további bemenetet, és mint az ábrán látható, az összes dinamikus szeletet előállítja kimenetként az adott tesztesethez, míg az utóbbinak további bemenetként meg kell adni a szeletelési kritérium maradék elemeit, az akciót és a használt változók halmazát, amely ez alapján kiszámítja a kritériumhoz tartozó egyetlen dinamikus szeletet.

A fejezet további alfejezeteiben megadjuk ezen fázisok megvalósításának részleteit C nyelvek analízisére.

7.3. Statikus analízis

A statikus analízis fázisának két szerepe van: elkészíteni a D/U programreprezentációt (7.4. alfejezet), valamint instrumentálni a forráskódot a végrehajtási nyom előállítása érdekében (7.5. alfejezet).

További feladata e fázisnak az algoritmusok által használt entitások belső azonosítását összerendelni az eredeti forráskód-sor számával. Ugyanis, az utasítás sorszámok egyedi számozásúak, és nem tükrözik az eredeti forráskód-sor számát, sőt egy utasítás több sorból is állhat és fordítva, egy sorban több utasítás is szerepelhet. Ehhez még hozzáadódik az, hogy a statikus analizátor előfeldolgozott fájlokkal dolgozik. Szeleteléskor viszont a felhasználói interfész az eredeti forrásra való hivatkozással kell hogy működjön. Ezért a statikus fázisban egy összerendelő táblázatot is generálunk, amely tartalmazza a kiszámított egyedi utasítás azonosítókat, hozzárendelve a megfelelő forráskód sorokhoz.



7.1. ábra. Az offline megvalósítás vázlata

A fenti feladatok elvégzéséhez egy teljes részletességű C analizátorra van szükség, amely képes szintaktikailag felismerni a bemeneti programkódot, és elemzés után előállítani valamilyen alkalmas belső reprezentációt. Ezt transzformálva megkapható a D/U reprezentáció, valamint a kigyűjtött információ alapján az eredeti forráskód megfelelő részeinek kiegészítése, instrumentálása. Erre a célra a dolgozat első részében ismertetett C/C++ forráskód-analizáló rendszert használtuk fel.

A 3. fejezetben ismertettük az analizátor front end kapcsolódási lehetőségeit. Az egyik az előállított ASG, a másik pedig a szintaktikus akciók interfésze. Az elsőből állítjuk elő a D/U programreprezentációt, amihez szükség van további származtatott információkra is. Ezeket a következő alfejezet megfelelő részeiben tárgyaljuk. Az instrumentáláshoz viszont a második lehetőséget használjuk fel, azaz egy kiegészített front end-et üzemeltetünk, amely-

ben a szintaktikus akciókat használjuk fel a megfelelő instrumentáló utasítások beszúrására a bemenet adott pontjaiban. Instrumentáláskor az előfeldolgozott forrásból származó, és a lexikális elemzés által előállított token-folyamot generáljuk vissza egy új forrásfájlba, közben kiegészítve azt a megfelelő szondákkal. Az instrumentáláshoz fel kell használnunk az addig felépült ASG-t is, hiszen abban találhatók meg a megfelelő szintaktikai információk (pl. blokk-határok, azonosítók nevei és típusai, stb.).

7.4. D/U programreprezentáció C-re

C nyelvre való megvalósításunkban egy kibővített D/U programreprezentációt állítunk elő, amit fájlban tárolunk a szeletelő algoritmusok számára. A 6. fejezetben e reprezentációt minden i utasításhoz az $i.\ d : U$ formában adtuk meg. Valós C programokra ezt kiegészítjük olymódon, hogy minden C utasításhoz $d :\ U$ elemek egy *sorozatát* rendeljük:

$$i.\ \langle (d_1 :\ U_1), (d_2 :\ U_2), \ldots, (d_{m_i} :\ U_{m_i}) \rangle\,,$$

vagy másképpen

$$DU^C = \left\langle \begin{array}{l} \langle (d_{11}, U_{11}), \ldots, (d_{1m_1}, U_{1m_1}) \rangle \\ \vdots \\ \langle (d_{I1}, U_{I1}), \ldots, (d_{Im_I}, U_{Im_I}) \rangle \end{array} \right\rangle$$

Használni fogjuk még a $DU^C(i) = \langle (d_{i1}, U_{i1}), (d_{i2}, U_{i2}), \ldots, (d_{im_i}, U_{im_i}) \rangle$ jelölést tetszőleges i utasításra.

E kibővítésre azért van szükség, mert egy C utasításban (ami gyakorlatilag egy mellékhatásokkal rendelkező kifejezés) több bal-érték is felvehet új értéket. A sorozatban az elemek sorrendje lényeges, hiszen korábbi elemek definiált változói felhasználhatók a későbbi használati halmazokban. A sorrendet a megfelelő részkifejezések kiértékelési sorrendje adja, ami pedig a C nyelv szabályaiból következik. Továbbá, az egyes használati halmazok is rendezettek kell hogy legyenek, mert egyes esetekben a feldolgozás sorrendje is lényeges, mint ahogy a kifejezések kiértékelési sorrendjét is adott esetben rögzíti a C nyelv. Erre a végrehajtási nyommal történő szinkron feldolgozás miatt van szükség (ld. például a mutató indirekciók és függvényhívások kezelését alább).

A D/U programreprezentáció további kiegészítése az, hogy a DU^C-ben szereplő változók nem egyszerűen skaláris és predikátum változók, hanem azoknak számos további jelentése lehet (a továbbiakban *változó* alatt nem csak a hagyományos értelemben vett programváltozókat értjük, hanem a különböző típusú virtuális változókat is):

1. *Skaláris változók*. Ezek a szokásos statikus tárolású (globális és lokális) programváltozók, amelyeknek állandó címük van a hívási veremben. A függvények formális paramétereit ugyancsak skalárisként kezeljük úgy, mintha azok a függvény lokális változói lennének.

2. *Predikátum változók*. Ezek az elvi algoritmus-leírásunkban elmondottaknak megfelelő virtuális változók. Jelölésük p_n, ahol n a predikátum-utasítás sorszáma. C esetében minden iterációs és szelekciós szerkezet predikátum változót fog eredményezni.

Még egy további speciális predikátum változó típust vezetünk be, amely segítségével a vezérlési függőségeket általánosan kezelhetjük. Ez az *entry* predikátum, jelölése $entry(f)$, és ezt minden függvényhez értelmezzük úgy, hogy az a függvény belépési pontjánál definiálódik és minden olyan utasítás használja, amely kívül esik egyéb strukturált predikátum függőségen (függvény-szintű láthatósági tartomány utasításai). Az *entry* predikátum a függvény elején definiálódik, és nincsenek használt változói.

3. *Kimeneti változók*. Jelölésük o_n, és ezek olyan virtuális változók, amelyeket olyan D/U elemeknél hozunk létre, ahol egy U használati halmaz szerepel, de nincs definiált változó. Ilyenek például a visszatérési értéket figyelmen kívül hagyó függvényhívások, mellékhatás nélküli kifejezés-utasítások, az ugró utasítások, és a jelen dolgozatban néhány „kimeneti" C utasítás a könnyebb érthetőség érdekében (pl. `printf`).

4. *Dereferencia változók*. Ezen virtuális változókat használjuk minden közvetlen memória cím, illetve az ott tárolt érték elérése esetén. Ez alapvetően a mutató indirekciókat foglalja magában, de ide tartozik még a tömb-indexelés és a struktúra-mezők elérése is. Jelölésük d_n, ahol n egy globális számláló az összes dereferencia előforduláshoz. Ilyen változókat az alábbi szerkezetekhez hozunk létre (minden esetben kivételt képez, ha a dereferencia struktúra objektumra vonatkozna, ekkor a mezők egyenként lesznek kezelve):

   - `*expr` *mutató típusú kifejezés indirekciója*
   - `object.member` *struktúra mezőjének elérése objektum-változóval*
   - `ptr->member` *struktúra mezőjének elérése mutatóval*
   - `array[index]` *tömb-indexelés*

   A dereferencia változók használatának lényege abban lesz, hogy a D/U reprezentációban csak szimbolikusan tüntetjük fel a függőségeket, majd a program futtatása során a végrehajtási nyomba ki lesz írva minden dereferenciához tartozó aktuális cím, amit majd a konkrét függőségek kiszámításánál behelyettesítünk a formális dereferencia változó helyére (részleteket ld. az algoritmusok megadásánál).

   Érdemes még megjegyezni továbbá, hogy a dereferencia változók sorrendje a használati halmazokban meg kell hogy egyezzen azzal a sorrenddel, ahogy végrehajtáskor találkozunk velük.

5. *Függvényhívás-argumentum változók*. Ezen virtuális változókat $arg(f, n)$-nel jelöljük, ahol f a hívott függvény neve és n az argumentum sorszáma. Minden függvényhíváshoz létrehozzuk ezen változókat az összes átadott argumentumhoz, és ezáltal összekötjük az argumentumot a formális paraméterrel. A változó definiálódik a hívás helyén (felhasználva az argumentumban szereplő további függőségeket), és használva lesz a hívott függvény belépési pontján, a megfelelő formális paraméter definiálásához. A függvény-mutatók kezelését később tárgyaljuk.

6. *Függvényhívás visszatérési érték változók*. Ezek $ret(f)$-fel jelölt virtuális változók, ahol f a visszatért függvény neve. Hasonlóan az argumentum változókhoz, ezeket is

létrehozzuk minden híváshoz, hogy összekössük a visszatérési értéket az érték felhasználásának helyével. A változó definiálódik minden kilépési pontnál (`return` utasítás), és használódik a hívás helyén lévő kifejezés használati halmazában.

A fenti kiegészített D/U programreprezentációban minden típusú változó azonosan kezelhető, például egy dereferencia változó függni fog egy predikátum változótól, ha a mutató indirekciót tartalmazó kifejezés vezérlési függőségben van az adott predikátum utasítással. Ilymódon, egy általános és tiszta ábrázolását kapjuk a C programok függőségeinek, amely rendkívül könnyű és biztonságos kezelhetőséget jelent a szeletelő algoritmusok számára.

Jelen alfejezet maradék részében áttekintjük, hogyan építjük fel a D/U reprezentációt a fenti típusú változókkal, tekintettel a C nyelv ide vonatkozó sajátosságaira. A kibővített D/U programreprezentációra láthatunk egy példát a 7.2. ábrán, amellyal a tárgyalt nyelvi sajátosságok nagy részét illusztráljuk.

i		$DU^C(i)$
	`#include <stdio.h>`	
	`int a, b;`	
1	`int f(int x,int y) {`	$entry(f) : \emptyset$, $x : \{arg(f,1)\}$, $y : \{arg(f,2)\}$
2	`  a += x;`	$a : \{entry(f), a, x\}$
3	`  b += y;`	$b : \{entry(f), b, y\}$
4	`  return x+2;`	$ret(f) : \{entry(f), x\}$
	`}`	
5	`void main() {`	$entry(main) : \emptyset$
	`  int s, *p;`	
6	`  s = 0;`	$s : \{entry(main)\}$
7	`  scanf("%d", &a);`	$a : \{entry(main)\}$
8	`  scanf("%d", &b);`	$b : \{entry(main)\}$
9	`  if (a < 0) {`	$p_9 : \{entry(main), a\}$
10	`  s = -1;`	$s : \{p_9\}$
11	`  goto label;`	$o_{11} : \{p_9\}$
12	`  s = 0;`	$s : \emptyset$
	`  }`	
13	`  p = &b;`	$p : \{entry(main), p_9\}$
14	`  while (*p < 10)`	$p_{14} : \{entry(main), p_9, p, d_1\}$
15	`  s += f(3,4);`	$arg(f,1) : \{p_{14}\}$, $arg(f,2) : \{p_{14}\}$, $s : \{p_{14}, s, ret(f)\}$
16	`  printf("%d", *p);`	$o_{16} : \{entry(main), p_9, p, d_2\}$
	`label:`	
17	`  printf("%d", s);`	$o_{17} : \{entry(main), s\}$
	`}`	

7.2. ábra. C példaprogram és annak D/U programreprezentációja

7.4.1. Adatfüggőségek

Az általunk alkalmazott D/U programreprezentáció a definiálás-használat adatfüggőségi kapcsolatokat – úgymond – *lokálisan* rögzíti minden utasításhoz. Ez azt jelenti, hogy a függő

(definiált) változóhoz csak a használt változók *neveit* rendeli hozzá, és nem a tényleges definiáló előfordulásokat, mint ahogy azt a hagyományos statikus függőségi analíziseknél tenni szokás (ld. például [39, 77]). Következésképpen a mi reprezentációnk felépítése nem igényel bonyolult függőségi vizsgálatokat, az megkapható a program egyszerű szintaxis-vezérelt feldolgozásából, követve minden egyes C kifejezés jelentését. A futáskor fellépő dinamikus függőségeket a D/U reprezentációból kiindulva határozzuk meg úgy, hogy az aktuális akciónál előbb meghatározzuk a változók által felvett memória címeket, majd a függőségeket ezen címek között rögzítjük.

Itt jegyezzük meg, hogy a konstans literálok értékül adása nem fog további függőséget eredményezni az értéket felvevő változók számára.

7.4.2. Függvényhívások

A függvényhívások és a paraméterátadás a D/U programreprezentációban a fent leírt arg és ret virtuális változók segítségével történik. Először is, minden függvényhívás kifejezéshez létrehozzuk az összes hívási argumentumhoz a megfelelő D/U elemeket, amelyekben az arg változók definiálódnak a megfelelő használati halmazokkal. Továbbá, minden függvény D/U reprezentációjában az összes formális paraméterhez létrehozunk egy D/U elemet, amelyben a paraméter van definiálva (mint egy további lokális skaláris változó), a használati halmaz pedig a megfelelő arg változót tartalmazza. Hasonlóan, minden függvény-visszatérési utasításhoz (és a visszatérés nélküli függvények utolsó utasítása után) egy D/U elem jön létre, a megfelelő ret definiált változóval és használati halmazzal. Végül, a hívási helyen ugyanezen ret változókat szerepeltetjük azon használati halmazokban, amelyek a hívást tartalmazó kifejezésekhez tartoznak. Ezek után, az interprocedurális függések a fenti virtuális változók, mint speciális globális változók révén valósulnak meg. Újból megjegyezzük, hogy az utasításokhoz tartozó D/U sorozatok és a használati halmazok elemeinek sorrendje a függvényhívások kezelése miatt is lényeges.

A visszatérés nélküli függvények kezeléséhez nem kötődik speciális tudnivaló, az ilyen függvények visszatérésekor csak további vezérlési függéseket hozhatnak be. A T. olvasó bizonyára hasonlóan könnyen el tudja képzelni a paraméter nélküli függvények és az alapértelmezett paraméterek kezelését. Függvénymutatók révén hívott függvények kezelése kissé speciális megoldást igényel, amit alább vázolunk a mutatók tárgyalásánál.

7.4.3. Vezérlési függőségek

Mint tudjuk, a D/U programreprezentáció a vezérlési függőségeket a (virtuális) predikátum változók használatával rögzíti. Ez úgy történik, hogy a hagyományos értelemben vett utasítások közötti *közvetlen* vezérlési függőségek kerülnek rögzítésre a megfelelő definiált és használati predikátum változók gyanánt, amint azt a 6. fejezetben leírtuk. Az általánosság érdekében minden függvényhez hozzárendeljük még az *entry* predikátumot a fent leírtak szerint.

Mint ahogy azt már hangsúlyoztuk, csak strukturált vezérlési szerkezetekkel rendelkező programok esetében a vezérlési függőségek könnyűszerrel meghatározhatók a program szintaktikai szerkezete alapján. C esetében ez megoldható lenne a szelekciós és az iterációs

szerkezetekre, azonban a *goto* feltétel nélküli ugrás és egyéb nemstrukturált vezérlések miatt bonyolultabb feldolgozást kell követnünk.

A vezérlési függőségek kezelésére az általános megközelítésből indulunk ki, mely szerint egy utasítás akkor függ egy predikátum utasítástól, ha valahányszor a predikátum végrehajtásra kerül, a függő utasítás végrehajtásának megtörténte csak a predikátum kiértékelésének eredményétől függ. A hagyományos definíció a program-függőségi gráfban (PDG) rögzíti a vezérlési függőségeket, pontos leírását számos helyen megtalálhatjuk, például [39, 77]. Ezt a megközelítést használják a statikus szeletelési algoritmusok is, például [54], hiszen az így kapott függőségek az összes lehetséges (statikusan kiszámítható, de bármely futásra érvényes) függőséget tartalmazzák. Maga a számítás a vezérlési folyam-gráfból (CFG) indul ki, és az ún. posztdominancia alapján határozza meg a függőségeket. A részletes algoritmus megtalálható az említett irodalomban.

A D/U programreprezentációnkhoz is a fenti módon határozzuk meg az összes potenciális vezérlési függőséget, és ezen információ alapján adjuk meg a bővített struktúrát az alábbiak szerint. Egész egyszerűen, ha egy i utasítás függ valamely másik utasítástól (amely ekkor predikátum kell hogy legyen), akkor az i-ben lévő használati halmazt bővítjük az adott predikátumhoz tartozó predikátum változóval. Mivel tetszőleges vezérlés-átadással rendelkező program esetén egy utasítás tetszőleges számú utasítástól függhet vezérlés szerint, ezek után a D/U-ban szereplő használati halmazokban is egynél több predikátum változó lehet (szemben a 6. fejezetben leírtakkal, ahol az elvi algoritmusaink pontosan egy predikátum változóval számoltak). Sőt, az is lehetséges, hogy egy utasítás nem is függ predikátumtól, amely esetben ez nem is elérhető és nyilván a szeletelésnél sem játszik szerepet.

Ha statikusan tekintjük ezen függőségeket, minden függőség potenciálisan megvalósulhat a program valamely futása során (megvalósulás alatt azt értjük, hogy az adott predikátum kiértékelődött és a függő utasítás is utána végrehajtódott, de ha a kiértékelés eredménye ellentétes lett volna, akkor a függő utasítás biztosan nem hajtódott volna végre az adott szituációban). Ugyanakkor, könnyen látható, hogy egy konkrét végrehajtáskor az aktuális utasításnak pontosan egy függősége lesz a megvalósult függőség. Ezt nevezzük az utasítás *aktív predikátumának*. Ha az aktuális utasítást és aktív predikátumát a végrehajtási sorszámaikkal együtt tekintjük, az így megkapott „dinamikus" vezérlési függést tekinthetjük a vezérlési függőség fogalmának kiterjesztését *akciókra*. (Megjegyezzük, hogy Korel másképp definiálja az akciók közötti vezérlési függőséget [64], a mi definíciónk természetesebb és lehetővé teszi a megvalósult függőségek követését.)

Az aktív predikátum meghatározását azon egyszerű elv szerint végezzük, hogy a potenciális vezérlési függőségek (használt predikátumok) közül kiválasztjuk azt, amely legutoljára lett definiálva. Ez alapján a szeletelő algoritmusaink az aktuálisan vizsgált utasítás függőségeinek továbbkövetésénél az összes használt predikátumból mindig kiválasztják azt az egyet, amellyel folytatni kell a számításokat. Más szóval, $i^j.\ d:\ U$ esetén azt a $p \in U$ predikátum változót választjuk, amelyre $LD(p) = \max\{LD(r) | r \in U$ és r predikátum változó$\}$, a j végrehajtási lépésben. Az LD értékeket könnyű nyilvántartani, hiszen amikor az i^j akciót hajtjuk végre, akkor $LD(d) = j$ értéket vesz fel, és ez az érték változatlan lesz egészen addig, amíg d egy későbbi lépésben újra definiálódik.

A fenti módszerrel egyaránt kezelhetők a *goto*, *continue*, *break* és *switch* nemstrukturált C szerkezetek, értelemszerűen felépített folyam-gráfból kiindulva.

A strukturált vezérlési szerkezetek közül a *do-while* érdemel egy kis további figyelmet. Itt

ugyanis a ciklusmagban szereplő utasítások potenciálisan a ciklus predikátumtól és a külső, tartalmazó strukturált predikátumtól is függnek. Ennek akkor lesz jelentősége, amikor a ciklus csak egyszer hajtódik végre, azaz a ciklus-predikátum első kiértékelése már hamis. Ebben az esetben a „külső" predikátum függőségeit kell követni. Azonban a fenti módszer ezt az esetet is helyesen kezeli.

Az alábbi példa C programrészlet jól szemlélteti a vezérlési függőségek kezelésének lényegét:

```
p1=x;
p2=y;
a=0;
if (p1==0) {
     a=1;
     goto c;
}
if (p2==0) {
     a=2;
  c: a=a+1;
}
b=a;
```

Tegyük fel, hogy az utolsó utasításban szereplő a változót adjuk meg a kritériumunkban, vagyis arra vagyunk kíváncsiak, hogy a b által felvett értéket mi befolyásolta. Háromféle futás lehetséges x és y értékeitől függően: az első predikátum lesz igaz (ekkor a második ki sem értékelődik), a második lesz az igaz, vagy egyik sem. Világos, hogy az első esetben x függőségeit kell tovább követni, a másodikban y-ét, a harmadikban egyikét sem, és pontosan ezt a viselkedést érjük el a fenti módszer alkalmazásával. Nevezetesen, az a=a+1 utasítás használati halmazában mindkét predikátum változó szerepelni fog, és a konkrét futás során derül ki, hogy melyik lesz az aktív predikátum: az első esetben p1, a másodikban p2 (a harmadikban nem is kerül a vezérlés az adott utasításra, ezért a függősége az első definíciójára fog vonatkozni).

7.4.4. Összetett bal-értékek

Némely C kifejezésnek az a mellékhatása, hogy a jobb oldali részkifejezés értékét felveszi a bal oldali részkifejezés, ekkor ezen részkifejezések valamely értékadó operátorral vannak összekapcsolva. Az ilyen kifejezések bal oldala módosítható bal-érték (l-value) kell hogy legyen, ami egyszerű esetben például egy változó. Viszont, összetett részkifejezésből is állhat egy bal-érték, amikor abban több változó és művelet is szerepel.

Ilyen utasításokhoz a D/U reprezentáció definiált változója továbbra is egyértelműen meghatározható, viszont nem triviális, hogy minek kell kerülnie a használati halmazba ami az adatfüggőségeket illeti. Például, egy *p típusú bal-érték esetén, ahol p valamilyen egyszerű adatra mutat, a használati halmazhoz fel kell-e vennünk magát a p mutatót? Ha szigorúan úgy tekintjük, hogy a definiált memória terület melyik más területektől függ (hiszen a szeletelő algoritmusok memória területekkel számolnak), a p által mutatott adat nem függ p értékétől. Ennek ellenére, mi mégis felvesszük p-t a függőségek közé. Ezt azzal

indokoljuk, hogy ha nem tennénk, a szeletből kihagynánk azokat az utasításokat, ahol p értéket kap, és következésképpen az is elveszne, hogy p rossz értéke miatt lesz rossz a definiált változó értéke. A fentiek alapján illusztrációnak megadunk néhány példát, ahol a D/U elemek adatfüggésekre vonatkozó részhalmazait is feltüntetjük:

```
a[i]   = r; //  d1:{r,i,a}
*(p+x) = r; //  d2:{r,p,x}
m.a    = r; //  d3:{r}
p->a   = r; //  d4:{r,p}
```

Megjegyezzük, hogy néhány statikus szeletelő megvalósítás is a fenti megközelítést alkalmazza.

7.4.5. Mutatók, mutató indirekciók, *címe* operátor és tömbök

A memória cella alapú megközelítésünk kézenfekvő megoldásokat kínál a mutatókkal kapcsolatos problémák kezelésére. Először is, ha egy mutató indirekció fel van használva egy kifejezésben, a kifejezésben értéket kapó változót függővé tesszük magától a mutatótól és a mutató által hivatkozott objektumtól is, nevezetesen a megfelelő dereferencia változótól. Ezt azzal indokoljuk, hogy a felvett érték nyilván függ attól is, hogy mely címet érjük el, de attól is, hogy mi található azon a címen. Ezt az esetet láthatjuk a példaprogram 14. sorában.

A fenti elv alól kivételt képez az az eset, amikor a mutató típusú (rész)kifejezés egy struktúra objektumra hivatkozik. Ekkor nem hozunk létre dereferencia változót, hiszen a struktúrát nem önálló egységként kezeljük, mert így nagyon konzervatív lenne a módszerünk. Ehelyett a struktúra minden tagjára egyenként meghatározzuk a dereferenciákat és a páronkénti függőségeket (struktúrák kezelését ld. lejjebb).

Amikor magával a mutatóval végzünk műveleteket, azt ugyanúgy kezeljük, mint minden más változót, kivéve, ha a mutatónak adunk értéket egy változó címének meghatározásával a *címe* operátor segítségével (&). Erre az a magyarázat, hogy az értékül vett cím nem függ az ott tárolt értéktől, képletesen egy konstans átadásáról van csupán szó (ld. példaprogram 13. sorát). Mutató által mutatott adat definiálásakor természetesen a dereferencia lesz a definiált változó.

Többszörös mutató indirekció esetén is a fenti szabályok alkalmazandók, nyilván minden egyes indirekció újabb dereferencia változó létrejöttét és a tőle való függőséget jelenti.

A tömbök kezelését könnyen megoldhatjuk, ha azt vesszük alapul, hogy C-ben a tömbök mutatóként is kezelhetők és minden tömb-elem elérés (indexelés) ekvivalens egy, az indexnek megfelelő eltolással megnövelt mutató indirekcióval. Ez alapján, tömbök esetében is ugyanúgy járhatunk el mint a mutatók kezelésénél, azzal a kiegészítéssel, hogy az indexben szereplő változó(ka)t is felvesszük a (használati) függőségek közé.

A függvénymutatók kezelése nem igényel lényegi átalakítást a bemutatott módszereinkben. Ugyanakkor, az algoritmusok leírásánál kihagytuk a kezelésük részleteit a könnyebb olvashatóság érdekében, ezért az alábbiakban ismertetjük ezeket. A legfőbb megoldandó probléma az, hogy mutatón keresztüli híváskor statikusan nem állapítható meg a hívott függvény, így az $arg(f, n)$ és $ret(f)$ virtuális változók sem. Helyettük, szimbolikus $arg(?, n)$ és $ret(?)$ változókat helyezünk el a D/U reprezentációban minden ilyen hívás esetén. Egyéb

szempontból, a mutatókra ismertetett D/U építési szabályok érvényesek a függvénymutatók esetében is.

Amikor a szeletelési algoritmusok eljutnak addig a pontig, hogy a hívást előkészítsék (argumentumok beállítása), még ezen ideiglenes változókkal számolnak, mígnem a végrehajtási nyom feldolgozásával eléri a hívott függvényt. Ekkor már ismert lesz annak neve is, így az adott híváshoz tartozó összes ideiglenes virtuális változóba behelyettesíti ezt a nevet (a globális algoritmusnál ez a hozzátartozó halmazok átnevezését, míg az igényvezéreltnél az EHT tábla hozzátartozó sorainak átnevezését/másolását jelenti). Az algoritmusok további végrehajtása a szokott módon történik. Későbbi, vagy egymásbaágyazott függvénymutatós hívások nem zavarják e módszert, hiszen minden pillanatban csak az aktuálisan feldolgozott híváshoz tartozó virtuális változók az ideiglenesek, és amint ismertté válik a hívott függvény, azonnal lecserélődnek mind az argumentum-, mint a visszatérési változók.

7.4.6. Struktúrák és union-ok

A C *union* típusú adatszerkezetek kezelését könnyen elintézhetjük, hiszen egy union-mező definiálásakor gyakorlatilag a többi mezőt is definiáljuk, sőt úgy is vehetjük, hogy az egész union objektumot is definiáljuk. Ezért a definiálás-használat viszonyokat is úgy rögzítjük, hogy bármely mezőre (vagy magára az objektumra) való hivatkozáskor az objektumot tekintjük mint skaláris változót. Ez azt jelenti, hogy a D/U reprezentációban skalárisként kezeljük azzal, hogy a mezőeléréseket lecseréljük magára az objektumra. Hasonlóan járunk el a *bitmezők* esetében is, hiszen ezen speciális mezőket nem tudjuk különállóan címezhető egységként kezelni.

A C struktúrák (*struct*) kezelése már összetettebb, hiszen itt szeretnénk megtartani a függőségek részletes követését mezőkre lebontva. Ha alacsony szinten tekintjük ezen adattípusokat, a mezők nem mások, mint a struktúra memóriában lévő kezdőcíméhez képest rögzített eltolású (offset) címen lévő adatok. Ilymódon, a struktúra kezelése hasonló a tömbökéhez, lévén hogy a szeletelő algoritmusok memória címeken dolgoznak. Konkrétan, minden mező eléréshez létrehozunk egy dereferencia változót, ami majd szeleteléskor különálló cellaként fog viselkedni, tehát a mezők közötti egyenkénti függőségeket tudjuk modellezni így. Hasonlóan, egy mező értékének használatakor a használati halmazba a dereferencia változó fog belekerülni.

A bonyodalmat az okozza, ha egy kifejezésben a struktúra változó van önmagában használva, amikor is gyakorlatilag az egész struktúrára az összes adattagjával együtt vonatkozik a művelet (struktúra másolás). Ekkor nem tehetünk mást, mint a műveletet modellezni az egyes mezőkre nézve, tehát struktúra-objektum értékadásakor, a függőségeket a mezőkre páronként meghatározzuk. Ez természetesen lehet rekurzív, a beágyazott struktúrák esetében minden nem struktúra al-mezőt külön fel kell sorolni.

Mivel maga a struktúra változó önmagában nem szerepel a függőségekben, a rá történő indirekciókat is át kell irányítani a mezőire. Ez azt jelenti, hogy nem hozunk létre és nem használunk dereferencia változókat ilyen esetben. (Mutatók kezelését ld. még az előző alfejezetben.)

Az alábbi kódrészletekkel illusztráljuk a dereferencia változók használatát struktúrák esetében (kommentben megadtuk a D/U elemek adatfüggésekre vonatkozó részhalmazait, valamint a mezőkre bontott függőségeket):

```
   struct S s,t,*p,*q;
   t.a = ...
   t.b = ...
1. q   = &t;   //  q : {}
2. s   = *q;   //    : {}
// s.a = q->a; // d2:{q,d1}
// s.b = q->b; // d4:{q,d3}
// ...
3. x   = s.a;  //  x : {d5}
4. p   = &s;   //  p : {}
5. y   = p->b; //  y : {p,d6}
```

A 3. sorban szereplő `x` használati halmaza `d5` elemének megfelelő végrehajtáskori cím (`s.a` mező) meg fog egyezni `d2` címével, ami végeredményben a `t.a`-tól való függést fogja eredményezni, és hasonlóan, `y` `t.b`-től fog függni.

7.4.7. Típuskényszerítés

A C típuskényszerítés (*type cast*) lehetővé teszi, hogy úgy hivatkozzunk adatokra, hogy azok eredeti típusa és értelmezése helyett más típusként érjük el azokat. Ezáltal ugyanazon tárolási területnek több különböző értelmezést adunk. A mi esetünkben ez akkor jelent problémát, amikor az adott tárterület különböző értelmezései más méretű adat-egységeket is jelentenek. Ekkor ugyanis a memória cella alapú szeletelési algoritmusaink nem fogják észrevenni az olyan függőségeket, amelyek az átlapolódó adatok definiálásából és használatából erednek. Például egy struktúra-mező definiálását a mező kezdőcímének definiálásaként értelmezzük, amely mezőt ha mindig „szabályosan", eredeti típusával érjük el, nem jelent problémát függetlenül az adott mező tárbeli hosszától. Ha viszont e struktúrát típuskényszerítéssel másképp érjük el, a hivatkozott tárterület más méretű lehet, más címen kezdődhet és tetszőleges részében átlapolódhat az eredetileg definiált adattal. Következésképpen, ha a típuskényszerített elérés más címen kezdődik, az alap algoritmusaink elveszítik a függőségeket. A probléma illusztrálására lássuk az alábbi példát:

```
struct S { int a;  int b;};
struct T {char c; char d;};
S *p = ...
p->a = 1;
x = ((T*)p)->d;
```

Itt elveszítjük az `x` használati halmazában a függőséget az <u>a</u> mező definiálásától egy sorral feljebb. És ilyen jellegű probléma jelentkezhet minden típuskényszerített mutatóval végzett művelet esetében. Ezt a problémát csakis úgy tudjuk áthidalni, hogy a hivatkozott memória cellák kezdőcíme mellett figyelembe vesszük az adott adattípus hosszát is, és a függőségek meghatározásánál a teljes címtartománnyal számolunk. Ezt a kiegészítést nem tüntettük fel az algoritmusaink leírásánál, hiszen túlságosan áttekinthetetlenné tenné a leírásokat. Továbbá, ha minden esetben címtartományokkal számolunk, ez ront az algoritmusok hatékonyságán, ugyanakkor csak ritka esetben jelent valódi pontatlanságot, ha a címtarto-

mányokat nem vesszük figyelembe, hanem csak a kezdőcímeket. Emiatt a megvalósításban is opcionális ez a funkció.

A megoldás részleteit az alábbiakban röviden ismertetjük. A D/U programreprezentáció a fent ismertetett módon épül fel, változtatás nélkül. Viszont, a végrehajtási nyomban a dereferencia jelölőket (amelyek az aktuális kezdőcímeket tartalmazzák) kiegészítjük a megfelelő kifejezés memóriabeli méretével (ez kiírható az instrumentált kóddal a `sizeof` operátor segítségével). Ezután, amikor a szeletelő algoritmusok a függőségeket követik, a megfelelő dereferencia változók függőségeit olymódon követik, hogy a kiírt tartomány *minden* bájton kezdődő címéhez felveszik a szükséges függőségeket. Ezáltal, ha a definiált terület bármely része a továbbiakban használva lesz, a függőségek helyesek lesznek. (Az instrumentálást, a végrehajtási nyomot és az algoritmusokat a következő alfejezetekben ismertetjük.)

7.5. Instrumentálás és a végrehajtási nyom

Ahogy azt a fejezet elején kifejtettük, offline megvalósításunkban forráskód-instrumentálást végzünk annak érdekében, hogy az így módosított program az eredetivel ekvivalens módon fusson ugyanazon lépéseket végrehajtva és eredményeket produkálva bármely tesztesetre, és eközben előállítsa az adott futáshoz tartozó *végrehajtási nyomot*. A nyom rögzíti a végrehajtott akciók egymás utáni sorrendjét, és még néhány egyéb információt a program dinamikus állapotairól, amelyek szükségesek a szeletelési algoritmusok számára. Ezek magukban foglalják a skalárisok, mutatók és dereferenciák dinamikusan felvett cím-értékeit, valamint a függvényhívásokhoz és lokális blokk határok eléréséhez tartozó információkat (ez szükséges a lokális változók láthatósági tartományának követéséhez). Az alábbiakban bemutatjuk a nyom szerkezetét, a benne tárolt információkat és előállításának módját.

7.5.1. A végrehajtási nyom fájlja

A végrehajtási nyom nem más, mint különféle típusú elemek lineáris sorozata, amely a program futásakor sorozatosan előáll, és a mi megvalósításunkban fájlban tárolódik. A sorozatnak ugyanakkor egy jól definiált szerkezete van, amely egy nagyon egyszerű és könnyen felismerhető nyelvtannal adott. A nyom fájl környezetfüggetlen nyelvtana a 7.3. ábrán látható.

A nyomban szereplő különféle elemek sorrendje adódik az instrumentált program végrehajtásakor bekövetkező és kiírandó események természetes sorrendjéből. Először az összes globális változóhoz tartozó adat kerül kiírásra a G jelölők révén, amelyek tartalmazzák a változó nevét és a hozzárendelt címet is. Ezután következik a program végrehajtásának nyomonkövetése, kezdve a *main* függvénnyel. Minden függvény belépéskor az FB jelölő kerül kiíratásra az adott függvény nevével, valamint kilépéskor FE generálódik.

A függvény törzsének végrehajtása közben háromféle instrumentáló tevékenység lehetséges: lokális változókhoz tartozó adat íródik ki hasonlóan a globálisokéhoz (D), beágyazott blokk íródik ki a függvénytörzzsel megegyező rekurzív tartalommal, vagy egy végrehajtott utasítás (akció) íródik ki. A blokkot a határolóival írjuk ki: belépéskor a BB jelölő generálódik a blokk egyedi azonosítójával, kilépéskor pedig BE, amely a kilépés után érvényben lévő blokk azonosítóját tartalmazza. A határolókat a szintaxis szerinti { és } zárójelek esetén és

$$
\begin{array}{rcl}
\langle trace\text{-}file \rangle & ::= & \{\, \langle global\text{-}var \rangle \,\}\ \langle main\text{-}function \rangle \\[1em]
\langle global\text{-}var \rangle & ::= & G\ (\ \mathbf{id}\ ,\ \mathbf{addr}\ ) \\[1em]
\langle local\text{-}var \rangle & ::= & D\ (\ \mathbf{id}\ ,\ \mathbf{addr}\ ) \\[1em]
\langle function \rangle & ::= & FB\ (\ \mathbf{id}\ )\ \{\, \langle function\text{-}body \rangle \,\}\ FE \\[1em]
\langle main\text{-}function \rangle & ::= & FB\ (\ \texttt{main}\ )\ \{\, \langle function\text{-}body \rangle \,\}\ FE \\[1em]
\langle function\text{-}body \rangle & ::= & \langle local\text{-}var \rangle \\
 & | & \langle action \rangle \\
 & | & \langle block\text{-}scope \rangle \\[1em]
\langle block\text{-}scope \rangle & ::= & BB\ (\ \mathbf{bnum}\ )\ \{\, \langle function\text{-}body \rangle \,\}\ BE\ (\ \mathbf{onum}\ ) \\[1em]
\langle action \rangle & ::= & E\ (\ \mathbf{inum}\ ,\ \mathbf{jnum}\ )\ \{\, \langle action\text{-}suffix \rangle \,\} \\[1em]
\langle action\text{-}suffix \rangle & ::= & P\ (\ \mathbf{addr}\ ) \\
 & | & \langle function \rangle
\end{array}
$$

7.3. ábra. A végrehajtási nyom formális szerkezete

minden nemstrukturált ugró utasítás esetében generáljuk, amikor is egy blokkot nemstrukturálatlanul elhagyunk és egy másikba ugyancsak nemstrukturálatlanul belépünk. Emiatt a belépő jelölő egy beágyazott blokk, a kilépő pedig általában a tartalmazó blokk, de ezek ugró utasítások esetében tetszőleges blokkot is jelenthetnek az aktuális függvényben.

Minden C utasítás (kifejezés) esetében *akció* jelölő generálódik, amely két részből áll. Az első rész (E) jelöli magát a végrehajtott utasítás i sorszámát és a j végrehajtási lépést (ez a hagyományos végrehajtási történet-beli elem). A második részben tetszőleges számú ún. *akció szuffix* generálódhat, amely az aktuális akcióhoz tartozó kiegészítő információkat tartalmazhat. Ez állhat egyrészt függvényhívásból, ha az akcióhoz tartozó kifejezésben van ilyen. Ez esetben a hívott függvény végrehajtásához tartozó teljes végrehajtási nyom kiíródik, rekurzívan.

A másik típusú akció szuffix a mutató indirekciók és egyéb memória elérések esetén generálódik. Ugyanazon esetekre, mint amikor a D/U reprezentációban dereferencia változókat hozunk létre (ld. feljebb) a nyomba is kiírjuk az elért memória címét a P dereferencia jelölő segítségével. Fontos megjegyezni, hogy minden D/U-beli dereferencia változónak meg kell hogy feleljen pontosan egy P jelölő, azonos sorrendben (amely a kiértékelés sorrendjétől függ), hiszen ezen sorrendiségre fognak építeni a szeletelő algoritmusok, amikor hozzárendelik a dereferencia változókat a konkrét címekhez.

7.5.2. Instrumentálás

Ahhoz, hogy a fenti nyom fájl tartalmát előállítsuk, az eredeti C forráskódot néhány helyen ki kell egészíteni – instrumentálni kell – a szükséges utasításokkal. Minden érintett helyre a kódban egy új függvényhívás kerül beszúrásra, amely végrehajtásával a megfelelő jelölők kerülnek kiírásra a nyom fájlba. Gyakorlati megfontolásból az instrumentált kódot C++ forrásként állítjuk elő, és nem kötjük magunkat a C viszonylag korlátozott lehetőségeihez. Ez számos tekintetben megkönnyíti az instrumentálást, és mivel a C++ nyelv gyakorlatilag magában foglalja a C-t, az eredeti forrás módosítás nélkül felhasználható (néhány apróság kivételt képez, például a régi típusú függvényparaméter deklarációk, de ezek könnyedén átírhatók szabályos C++ kóddá). Könnyebb például néhány instrumentáló függvényt C++ sablon (*template*) függvényként megvalósítani, és utasításokat szúrhatunk be a lokális változók függvény elején lévő deklarációja elé is. A tényleges instrumentáló függvények megvalósítása egy kiegészítő fejléc- és implementációs fájlban van megadva, amelyek az instrumentált forrás fordításakor a programhoz hozzászerkesztődnek (ezen fájlok néhány részlete látható a 7.4. ábrán).

Az alábbi felsorolásban összegezzük, hogyan történik az instrumentálás a végrehajtási nyom egyes elemeinek esetében:

1. A függvény- és blokk-határoló jelölők könnyedén generálhatók úgy, hogy elhelyezzük a megfelelő instrumentáló függvényhívásokat a függvény/blokk elejére és végére. Az egyedi blokk azonosítók a statikus fázisban generálódnak. Meg kell jegyezni, hogy a nemstrukturált ugrások is blokk kezdést és befejezést eredményeznek, ezért ezekhez is legyártjuk a megfelelő jelölőket. Hasonlóan, a függvények végét minden *return* utasítás előtt jegyezni kell. Az eredetileg saját blokk nélküli, szelekciós- vagy iterációs utasítások egyetlen törzs-utasításához új blokk is generálódhat.

2. A lokális és globális változók futáskor felvett memória címeit a *címe* (&) operátor segítségével íratjuk ki, a lokálisokat közvetlenül a deklarációjuk után, a globálisokat pedig a *main* függvény legelső utasítása előtt.

3. Minden kifejezéshez egy akció jelölőt kell generálni. Ezt azzal az egyszerű trükkel végezzük, hogy az instrumentáló függvényhívást a kifejezés elé helyezzük, majd összekötjük azt a kifejezéssel a *vessző* operátor segítségével. Ezáltal megtörténik a függvényhívás, majd utána a kifejezés kiértékelése is, és a kiegészített kifejezés értéke is helyes lesz. Az i utasítás sorszám sorozatosan számítódik ki a statikus analízis fázisában, a j lépés sorszámhoz pedig egy számlálót generálunk az instrumentált forráskódba, amely minden utasítás végrehajtáskor automatikusan inkrementálódik.

4. A dereferencia jelölőket egy C++ sablon instrumentáló függvény hívásával hozzuk létre, amely egyszerűen kiírja a nyomba a paraméterül kapott cím-értéket és visszaadja ugyanazt, hogy megmaradjon az eredeti viselkedés. A függvény a kapott paramétert egy mutatóként veszi át, amely típusa sablon-paraméterként kerül átadásra, mindez annak érdekében, hogy az eredeti kifejezés típusa megmaradjon a régi.

Hogy a T. olvasónak legyen rálátása a konkrét instrumentálási programkódokra, a 7.4. ábrán megadunk néhány példát erre vonatkozóan.

```
Int32 nb, na, mid;
nb = 0;
na = 256;
do {
   mid = (nb + na) >> 1;
   if (indx >= cftab[mid]) nb = mid; else na = mid;
}
```

```
 int nb , na , mid ;
D_VA(&nb,"nb");
D_VA(&na,"na");
D_VA(&mid,"mid");
D_EH( 1259,D__ec++); D_EB() ,(nb = 0);
D_EH( 1260,D__ec++); D_EB() ,(na = 256);
 do
{ /*BlockGuard*/

    {
    D_SB(243);
     D_EH( 1262,D__ec++); D_EB() ,(mid = (nb+na)>>1);
    D_EH( 1263,D__ec++);  if ( D_EB() ,(indx>=(*D_P(&cftab[(mid)]))) )
    { /*BlockGuard*/
    D_EH( 1264,D__ec++); D_EB() ,(nb = mid)
    ;} /*BlockGuard*/

    else
    { /*BlockGuard*/
    D_EH( 1265,D__ec++); D_EB() ,(na = mid)
    ;} /*BlockGuard*/
;
    D_SC(242);
     }

;} /*BlockGuard*/
```

```
template <class T>
T* D_P(T* p)
{
  D_STRM << "P(" << (void*)p << ")\n";
  return p;
};

void D_SB(int b)
{
  D_STRM << "SB(" << b << ")\n";
}

void D_EH(int i, int j)
{
  D_STRM << "E(" << i << "," << j << ")\n";
}
```

7.4. ábra. Az eredeti és az instrumentált forráskód, valamint néhány instrumentáló függvény

7.6. Globális algoritmus C-re

A globális dinamikus szeletelési algoritmus C-re kiegészített változata a 7.5–7.7. ábrákon látható. A pszeudokódban a $TR \gg tr$ jelölést használjuk a következő elem (tr jelölő) kiolvasására a TR végrehajtási nyomból, amit ilymódon jelölők (és a hozzá tartozó kiegészítő információk) folyamának tekintünk (ld. előző alfejezetet). Továbbá, egy S halmaz e elemmel való bővítését az $S \leftarrow e$ jelöléssel illetjük. A többi jelölés magától értetődő.

Az algoritmus főprogramja a GlobalisAlgoritmusCre program, melynek két bemeneti paramétere van: a P program, amelyet szeletelünk és annak egy $\mathbf{x}$ bemenete, amelyhez tartozó tesztesethez állítjuk elő az összes dinamikus szeletet. Első lépésként rögzítjük a végrehajtási nyomot, amelyet az algoritmus során sorozatosan olvasunk az elejétől kezdve. Az algoritmus vezérlése kettős: elsődlegesen a nyom szintaktikai szerkezetét követi, másrészt a D/U programreprezentáció elemeit is folyamatosan dolgozza fel. Ezért a nyom és a D/U összhangban kell hogy legyen, például a függvényhívások és dereferenciák szinkron kezelése miatt. Ha valahol sérül a szinkronizáció (amely normál esetben nem következhet be), az algoritmus $SzinkronizaciosHiba$ üzenettel leáll. Az algoritmus egy függvény egy időben történő feldolgozása köré szerveződik, mégpedig olyan sorrendben, ahogy az a végrehajtáskor történik, tehát ahogy a nyom rekurzív szerkezete azt előírja. Ezzel párhuzamosan működik a program a FuggvenyFeldolgozas eljárás rekurzív hívásával, tehát a főprogram is a `main` függvény feldolgozásával kezdődik, a globális változók címeinek feldolgozása után.

A feldolgozás közben szükség van egy segéd struktúrára, amely a globális és lokális változók láthatósági információit tárolja. E struktúra, melyet sc-vel jelölünk egy verem, amelyre a futás közben elért és elhagyott láthatósági tartományok (*scope*-ok) kerülnek. A verem a változók aktuális címeinek tárolására szolgál, a különböző példányok megkülönböztetése érdekében. Erre két oknál fogva van szükség: először is, függvények rekurzív hívásakor a statikus tárolású változóknak a hívási vermen különböző példányaik lesznek, különböző címekkel. A másik ok az, hogy egyes újabb C implementációkban azonos nevű változók használhatók különböző láthatósági tartományokban, tehát nem kötelező azok deklarációja a függvény elején. E verem a FuggvenyFeldolgozas eljárásban van karbantartva, ahogy azt a végrehajtási nyom diktálja. Először is, a verem tetején létrehozunk egy tartományt, amint egy függvénybe belépünk (FB jelölő). Ezen kívül a blokkoknak is létrehozunk új tartományokat belépéskor (BB), kivéve ha az adott blokk már létezik a vermen (ez nemstrukturált vezérlésátadáskor történhet meg). Utóbbi esetben csak egy mutatót állítunk rá az aktuális tartományra. Mivel az aktuális függvényen belül bármely blokkba lehetséges a nemstrukturált ugrás, a blokkokat nem törölhetjük a veremről kilépéskor (BE jelölő esetén), csak az aktuális tartomány mutatóját állítjuk át. A blokkok törlése a függvényük törlésével egyidejűleg történik kilépéskor, tehát a FE jelölő elérése esetén.

A FuggvenyFeldolgozas eljárás további feladatai a lokális változók címeinek tárolása a vermen (D jelölő beolvasása és a verem tetején lévő aktuális tartományhoz adása), valamint a végrehajtható utasítások feldolgozása, amely nem más, mint a tényleges függőségszámítás a végrehajtott akciókon (AkcioFeldolgozas eljárás).

Az AkcioFeldolgozas eljárás a paraméterül kapott i^j akcióhoz tartozó dinamikus függőségek kiszámítását végzi a megfelelő memória-cellákon, és a hozzátartozó dinamikus szeleteket a kimenetre adja a következő módon (itt feltesszük, hogy a kritériumban szereplő változóhalmaz megegyezik az utasítás teljes használati halmazával). Az i utasításhoz tartozó DU^C

elemeket sorban feldolgozzuk kezdve az elsővel (a sorrend számít!), és minden elemhez kiszámoljuk a hozzátartozó $i.\ d'_k : U'_k$, ún. *dinamikus D/U elemet*. Ez az eredeti valós és virtuális változók helyett már a memória cellák közötti függőségeket rögzíti. Végül, ezek alapján meghatározzuk a szokásos $DynDep$ halmazokat, valamint a hozzátartozó LS és LD értékeket (ez utóbbiak a predikátum változókhoz szükségesek). Az LD értékek nyilvántartása az LS értékekkel párhuzamosan történik, ügyelve a műveletek sorrendjére, amint azt a 6. fejezetben már láttuk. Mint ahogy a végrehajtási nyom két típusú akció szuffix (függvényhívás és dereferencia) sorozatát is tartalmazhatja minden akcióhoz, az AkcioFeldolgozas eljárásnak e két szuffix beolvasását is el kell végeznie, ahogy azt az alábbiakban látni fogjuk.

Egy dinamikus D/U elemhez a következő számításokat kell elvégezni. Először minden statikus használati változót, majd a definiált statikus változó címét is *feloldjuk* a CimFeloldas eljárás hívásával. Feloldáskor a statikus változókhoz meghatározzuk a dinamikus memória cellákat, amelyeket a számításoknál kell felhasználni. Ez a skalárisok és dereferenciák, az aktuális, j-edik lépésben érvényes címeinek kikeresését jelenti. A skalárisok esetében a keresés a tartomány-verem tetejéből kiindulva történik, a szokásos láthatósági szabályoknak megfelelően. A dereferencia változók által felvett címeket pedig a végrehajtási nyomból olvassuk be (P akció szuffix), amelyek a D/U reprezentációnak megfelelő sorrendben kerültek kiírásra az instrumentálás során. Feloldás nem szükséges az összes egyéb típusú változó (például predikátumok) esetén.

Egy akció feldolgozásakor el kell még végezni a vezérlési függőségek megfelelő kezelését is, ahogy azt a 7.4. alfejezetben bevezettük. Nevezetesen, az U_k halmazból kiválasztjuk azt az aktív predikátumot, amely legutoljára lett definiálva: a PR halmazba összegyűjtjük az összes statikus predikátumot, majd a legnagyobb LD értékkel rendelkezővel bővítjük a dinamikus U'_k halmazt. Megjegyezzük, hogy minden végrehajtott utasításhoz így pontosan egy aktív predikátumot fogunk rendelni.

Az AkcioFeldolgozas eljárásban történik még a függvényhívások kezelése is az alábbiak szerint. Minden esetben amikor a D/U reprezentációban egy függvényhívás visszatérési érték virtuális változó (ret) kerül sorra a használati halmaz feldolgozásakor, a FuggvenyFeldolgozas eljárás rekurzív hívásával a hivatkozott függvény teljes feldolgozását is elvégezzük (a nyomban FB akció szuffix kell hogy következzen). Ez azelőtt történik mielőtt még a jelen akció további használati függőségeit feldolgoznánk, hiszen így tudjuk meghatározni a felhasználni kívánt ret változót. A szeletelő program ilyen jellegű vezérlése pontosan meg fog felelni az analizálandó program végrehajtásának és így a kiírt végrehajtási nyom szerkezetének is.

7.6.1. Komplexitási tényezők

A globális dinamikus szeletelési algoritmus komplexitása a következő többlet-tényezőket hozza C-re az elvi leíráshoz képest: a láthatósági tartomány verem karbantartása és benne való keresés, a dereferencia változók kezelése és memória cellákon történő szeletelés, valamint az aktív predikátum meghatározása. A többi C különlegesség, amiket a fejezet elején felsoroltunk, nem jelent lényeges többletet.

Az aktív predikátum meghatározásához az LD értékek tárolása szükséges, de ez nem jelentős, hiszen csak a predikátumokhoz kell egy-egy egész számot tárolni. Továbbá, a PR

halmaz gyakorlatilag elhagyható az implementációból, hiszen csak egy maximális értéket kell nyilvántartani.

A tartomány-verem mélységét a végrehajtás hívási verem mélysége és a függvények vezérlésátadási (blokk-) szerkezete határozza meg. Továbbá, az egyes szinteken lévő elemek számát a globális és lokális változók száma adja. Ezen tényezők határozzák meg a veremben való keresés műveletigényét is, amit a skalárisok címének feloldásakor kell elvégezni. Ugyanakkor, ne feledjük, hogy a blokk-szerkezetet nem kell nyilvántartanunk ha olyan C programot analizálunk, amelyben minden lokális változó a függvény elején deklarálva van. Ekkor a keresés gyakorlatilag konstans idejű lehet, hiszen mindig a verem tetején lévő függvénytartományt kell használni, a változók címeit pedig tárolhatjuk tömbben.

A memória cellákon történő szeletelés azt vonja maga után, hogy a kezelt $DynDep$ halmazok és LS értékek számát a használt memória cellák (kezdőcímek) száma határozza meg (kiegészítve egyéb, például predikátum változókkal).[2] Ez a szükséges dereferencia változók cím-feloldásainak számát is jelenti, ami pedig nem más, mint a végrehajtási nyom P jelölőinek feldolgozása. E szám természetesen nagyban függ az analizálandó program jellegétől. Például egy tömböket erőteljesen használó programban sok memória indirekció lehet. Méréseink ugyanakkor azt igazolják (ld. a 8. fejezetet), hogy a használt memória címek átlagos száma és a program mérete között nagyobb mértékű összefüggés fedezhető fel, mint a végrehajtási lépésszámmal összehasonlítva. Sőt, a címek száma csak nagyon kis mértékben tér el a (statikusan ismert) skalárisok és predikátumok számától, azon belül is a skalárisok dereferenciái vannak döntő többségben.

Érdemes még megjegyezni, hogy az algoritmus végrehajtása során szükség van tetszőleges (címére feloldott) változóhoz tartozó $DynDep$ halmaz és LS érték elérésére. Míg ez a művelet az elvi algoritmusnál konstans időt vesz igénybe (a különböző változók száma statikusan ismert, így ezeket sorszámozva egy tömbön keresztül lehet kezelni), addig a memória cellákon történő szeletelés esetében a kiválasztás valamilyen asszociatív tárolón keresztül történhet csak, hiszen a különböző változók száma (és „neve", ami nem más mint a cím) dinamikusan lesz csak ismert. Itt viszont általában logaritmikus keresést kell végezni a változók számának függvényében, ami a komplexitás lineáris tényezőjét növeli még egy $\log(V)$)-vel, a definiált különböző memória címek számának függvényében.

Továbbá, vegyük észre, hogy a dereferenciák címének feloldását optimálisabban is meg lehet oldani az implementációban. Nevezetesen, a hivatkozott címet csak olyan esetekben kellene kiírni, amikor az különbözik a korábban felvett értéktől (túlnyomórészt nem ez az eset áll fenn), maga a feloldás pedig működhetne egy tömbön, amelyben a változók legutóbbi címeik vannak tárolva és szükség esetén módosítva.

7.7. Igényvezérelt algoritmus C-re

Az igényvezérelt dinamikus szeletelési algoritmusunk C nyelvre történő kibővítéséhez szükségünk van még egy további segéd struktúrára, amit *cím-történet táblának* (AHT vagy execution history table) fogunk nevezni. E táblát arra fogjuk használni, hogy kikeressük belőle a változók aktuális címeit (skalárisok címei és dereferencia változók értékei) egy adott végrehajtási lépés pillanatában. A globális algoritmusunkban nincs szükség ilyen segéd táb-

[2] A típuskényszerítés teljes kezelése esetén ez a memória-területek hosszától is függ, ld. feljebb.

lára, hiszen a kérdéses címek minden lépésben pontosan az aktuális címek lesznek, lévén hogy a végrehajtási nyommal párhuzamosan dolgozik az algoritmus. Ezzel szemben, az igényvezérelt módszerünkben a nyomot fordított irányban dolgozzuk fel, azaz egy adott iterációban feldolgozott akció után tetszőleges korábbi lépéssel folytatódhat az algoritmus. Emiatt szükség lehet a dereferenciák tetszőleges lépésnél felvett értékeire. Szerencsére, egy dereferencia által felvett cím általában nem változik lépésről lépésre, ezért az AHT táblában a változók által felvett címeket csak változás esetén rögzítjük, feltüntetve a lépés sorszámot, ahol a változás történt:

$$\begin{array}{c|llll}
a_1 & (j, addr)_{a_1 1} & \cdots & (j, addr)_{a_1 m_{a_1}}, & j_{a_1 1} < \ldots < j_{a_1 m_{a_1}} \\
\vdots & & & & \\
a_N & (j, addr)_{a_N 1} & \cdots & (j, addr)_{a_N m_{a_N}}, & j_{a_N 1} < \ldots < j_{a_N m_{a_N}} \\
\hline
d_1 & (j, addr)_{d_1 1} & \cdots & (j, addr)_{d_1 m_{d_1}}, & j_{d_1 1} < \ldots < j_{d_1 m_{d_1}} \\
\vdots & & & & \\
d_M & (j, addr)_{d_M 1} & \cdots & (j, addr)_{d_M m_{d_M}}, & j_{d_M 1} < \ldots < j_{d_M m_{d_M}}
\end{array},$$

ahol N a különböző skaláris-, és M a különböző dereferencia változók száma, amelyek használva vannak a végrehajtás során. A tábla soraiban felsoroljuk a skaláris (a) és dereferencia (d) változókat, és az egyes változókhoz a soraikban $(j, addr)$ lépés-cím párokat listázunk. Ha egy v változó AHT bejegyzése $(j, addr)$, ez azt jelenti, hogy a változó a j-edik lépésben az $addr$ címet vette fel, és ez az érték az is maradt egészen valamely későbbi $j' > j$ lépésig. Mivel a táblában szükség lesz keresésre bármely lépés-sorszám alapján, érdemes definiálnunk az $AHT^{<}(v, j)$ függvényt, amely visszaadja a v sorában szereplő, j-hez tartozó címet, ha létezik ilyen. Ha nem, akkor a legnagyobb j előtti címet ugyanabban a sorban. Az $AHT^{<}$ függvényben a keresést könnyű logaritmikus időben implementálni, hiszen a tábla-bejegyzések rendezetten vannak felsorolva.

Ezen kívül, vegyük észre, hogy az EHT táblát is kissé másképp kell értelmeznünk a memória cella alapú szeletelés miatt. Nevezetesen, a tábla sorait a ténylegesen definiált (dinamikus) változók határozzák meg, ami nem más mint a használt memória címek és egyéb virtuális változók. Ennek az lesz a következménye, hogy minden művelet e táblán (építés és keresés) a tábla sorának azonosítása miatt költségesebb lesz, hiszen itt már asszociatív tárolót vagyunk kénytelenek használni. Tudniillik, a konkrét változók, amelyek a sorokat alkothatják statikusan nem ismertek és nem lehet azokat sorszámozással például tömbben tárolni.

Felhasználva ezen kiegészítéseket, megadhatjuk az igényvezérelt algoritmust C nyelvekre, ami nagyon hasonló lesz az elvi módszerhez, a szükséges módosításokkal. Az algoritmus a 7.8. és 7.9. ábrákon látható (a pszeudokódban használt jelölések megegyeznek a globális algoritmusnál ismertetettekkel).

A főprogram az IgenyvezereltAlgoritmusCre nevű program, amelynek bemenete az analizálandó P program és a C dinamikus szeletelési kritérium, és a program ez alapján előállítja a megfelelő dinamikus szeletet. A könnyebb érthetőség végett itt is feltesszük, hogy a kritériumban szereplő változó-halmaz megegyezik az utasítás teljes használati halmazával. Első lépésként rögzíteni kell a végrehajtási nyomot a kritériumban szereplő akcióval bezárólag (ld. még a lábjegyzetet az elvi leírásunkhoz a 6. fejezetben). A nyom EHT formátumúra hozását a TablakKiszamitasa eljárásba tettük, de ez a művelet elvégezhető a végrehajtás-

sal párhuzamosan is, azaz a lineáris alak külön tárolására természetesen nincs is szükség. Továbbá, ugyanez az eljárás felelős az AHT segéd struktúra felépítéséért is.

A táblák felépítése után a főprogram szerkezete azonos az elvi algoritmusunkéhoz. Nevezetesen, a $worklist$ munka-tárolót használjuk a még feldolgozásra váró akciók nyilvántartására, amelyből iterációnként egyet kiveszünk (kezdve a kritérium akciójával), ahhoz megkeressük a memória címre feloldott használati változók utolsó definíciót, majd ezekkel bővítjük a $worklist$-et. Az algoritmus akkor ér véget, amikor ez kiürül, és az összes közben feldolgozott akció utasítása adja a dinamikus szeletet.

Az első kiegészítése az algoritmusnak az, hogy az aktuális iteráció k utasításához minden hozzá tartozó DU^C elemet fel kell dolgozni. Továbbá, minden u_{mn} használati változónak előbb meghatározzuk a dinamikus megfelelőjét (u'_{mn}), majd azzal követjük tovább a függőségeket. A dinamikus függőségeknek hasonló értelmezése van, mint a globális algoritmusnál: skaláris és dereferencia változók esetén a feldolgozás alatt álló l lépés sorszámnak megfelelő címet jelenti (ezt az AHT táblából keressük ki), egyébként meg a statikus használati változót, és így érjük el a használt memória cellákon történő szeletelést. Ezután a dinamikus használati változóknak megkeressük a korábbi legutolsó definícióit az $EHT^<$ függvény segítségével, és ezekkel bővítjük a $worklist$-et. Ez alól a predikátum változók kezelése a kivétel, hiszen itt is szükség van az aktív predikátum meghatározására a nemstrukturált vezérlések miatt. Ezt úgy tesszük, hogy először az aktuális U_m használati halmazban szereplő összes predikátum változónak meghatározzuk az $EHT^<$ értékét, ezeket összegyűjtjük a PR halmazban, amelyből végül kiválasztjuk a legnagyobb lépés-sorszámmal rendelkezőt, majd azzal az egyel bővítjük a $worklist$-et.

A TablakKiszamitasa nevű eljárás szolgál a végrehajtási- és cím-történet táblák (EHT és AHT) kiszámítására a szeletelő algoritmus végrehajtása előtt. Itt a lineáris végrehajtási nyomot dolgozzuk fel (akár a program futásával egyidejűleg), nagyon hasonló módon, mint ahogy a globális szeletelő algoritmusunk teszi a függőségi halmazok számítását (ld. a 7.5. és 7.6. ábrákat). Konkrétan, a TablakKiszamitasa a globális algoritmus főprogramjával egyezik meg, amely a nyom beolvasását kezdi el a globális változók és a `main` függvény feldolgozásával. A FuggvenyFeldolgozas eljárás is ugyanazon tevékenységeket végzi el, viszont egy akció feldolgozásakor nem a függőségi halmazokat számoljuk, hanem az AkcioFeldolgozas eljárást a táblák építésére használjuk fel. Ez, a függvényhívás szokásos kezelése mellett, a paraméterben kapott akció definiált változójának címét feloldja (a CimFeloldas eljárás is változatlan), majd az EHT megfelelő sorát bővíti ugyanazzal az akcióval. Ezen kívül az adott utasítás összes skaláris és dereferencia használati változójának címét is feloldja, majd a megfelelő AHT sorokat bővíti a kapott címekkel, feltéve hogy azok mások mint az utolsó bejegyzések.

7.7.1. Komplexitási tényezők

Az igényvezérelt dinamikus szeletelési algoritmus komplexitását az alábbi többlet-költségek terhelik a C nyelv sajátosságainak kezelése miatt.

A táblák építésének fázisában a tartomány-verem kezelése a már ismertetett többletet jelenti. Továbbá, az építés maga után vonja, hogy az összes végrehajtott akcióhoz megvizsgáljuk a DU^C-ben szereplő összes változót, és elvégezzük a szükséges feloldásokat. Ezek mennyisége és komplexitása megegyezik a globális algoritmusnál ismertetettekkel. Ami to-

vábbi többletet jelent, az az, hogy az igényvezérelt algoritmus főprogramjában újból meg kell vizsgálni a változókat, igaz, itt csak a függőségekben szereplő akciókét és csak a használatikat. Viszont, itt már nem kell a feloldást elvégezni, hanem a táblákban keresni. Maga a táblabővítés az AHT esetében konstans időt vesz igénybe, viszont az EHT-nél ez már logaritmikus művelet, mert a tábla sorát előbb meg kell keresni a cím alapján, amelyeknek száma (és keresési kulcsa) statikusan nem ismert, ezért asszociatív tárolóra van szükség.

A főprogramban az aktív predikátum kezelése – hasonlóan a globális algoritmushoz – nem jelentős többletköltség. Azonban, az AHT struktúra szükségessége jelent némi hozzáadott komplexitást. E táblának annyi sora van, ahány különböző skaláris és dereferencia változó kerül használatra a végrehajtás során. Nyilván ez kevesebb, mint az összes változók száma, de mindenféleképpen az analizálandó program jellegétől függ. E szám általában a program méretével van (lineáris) összefüggésben (ld. méréseinket a következő fejezetben), sőt méréseink azt is igazolják, hogy a skalárisok és dereferenciák aránya is állandónak tűnik az összes változóhoz képest. Egy sor hossza az adott változó címének váltakozási gyakoriságától függ. Ahogy az várható, és a méréseink is ezt igazolják, ezek viszonylag kis számok a végrehajtás hosszához képest, méréseink szerint az átlag értékek minden esetben skalárisoknál 10, dereferenciáknál 60 alattiak voltak (milliós nagyságrendű végrehajtásunk is volt). A táblában való keresés komplexitását a sor kiválasztása és a soron belüli keresés határozza meg. Az előbbi konstans idejű, míg az utóbbi logaritmikus a sor hosszának függvényében.

Végül, a főprogramban az EHT táblában való keresést két tényező befolyásolja: az elvi leírásunknál már tárgyalt sorban való keresés (logaritmikus) és a tábla sorának kiválasztása, amely szintén logaritmikus C esetében, mint ahogy azt fent már említettük.

Az algoritmus táblaépítés részét nem tekintve, a fenti két keresés a műveleti komplexitás növekedését fogja előidézni, mégpedig a logaritmikus tényezőben egy $\log(V)$-vel.

7.8. Összegzés

Láttuk, hogy a szeletelési algoritmusaink elvei alapvetően rendkívül egyszerűek. Mégis, amint egy valós programozási nyelvet célzunk meg, a problémák megsokszorozódnak. E fejezetben ismertettük, hogyan oldottuk meg a C nyelv összes specialitásának kezelését az algoritmusainkban. Láthatjuk, hogy a hozzáadott feldolgozások némi hátrányt jelentenek a hatékonyságot illetően, de a módszerek így is sokkal jobban alkalmazhatók a gyakorlatban, mint a korábbi módszerek. A következő fejezetben ezt a méréseink eredményeivel is alátámasztjuk.

```
program  GlobalisAlgoritmusCre(P, x)
input:   P : program
         x : program bemenet
output:  dinamikus szeletek minden (x, i^j, V_i) kritériumhoz
         (j = 1 ... J, V_i = DU^C(i) használati halmazainak uniója)
global:  tr : végrehajtási nyom eleme
         sc : láthatósági tartomány verem
begin
  TR végrehajtási nyom rögzítése
  TR >> tr
  Globális tartomány létrehozása sc tetején
  while tr = G(id, addr)
    id tárolása addr címmel sc tetején
    TR >> tr
  endwhile
  if tr ≠ FB(main) then SzinkronizaciosHiba
  else FuggvenyFeldolgozas(main) endif
  if tr ≠ FileVege then SzinkronizaciosHiba endif
end
```

7.5. ábra. A globális algoritmus C-re (főprogram)

```
procedure FuggvenyFeldolgozas(f)
begin
  while tr ≠ FE és tr ≠ FileVege
    case tr of
      D(id, addr) :
        id tárolása addr címmel sc tetején
      FB(f) :
        Új függvény tartomány létrehozása sc tetején
      BB(block-num) :
        block-num sorszámú blokk tartomány létrehozása az aktuális függvényhez
          sc-ben, ha az még nem létezik, és az aktuális tartomány beállítása
      BE(outer-num) :
        Az aktuális tartomány mutatójának sc-ben outer-num-ra állítása
      E(i, j) :
        AkcioFeldolgozas(i, j)
    endcase
    TR ≫ tr
  endwhile
  if tr = FE then
    A függvény- és a hozzá tartozó összes blokk tartomány törlése sc tetejéről
    TR ≫ tr
  else SzinkronizaciosHiba endif
end
procedure CimFeloldas(x)
begin
  case x típusa of
    skaláris:
      return x' := x megkeresése sc-ben és címének visszaadása
    dereferencia:
      if tr ≠ P(addr) then SzinkronizaciosHiba
      else
        TR ≫ tr
        return x' := addr
      endif
    minden egyéb virtuális:
      return x' := x
  endcase
end
```

7.6. ábra. A globális algoritmus C-re (függvény-feldolgozás és címfeloldás)

```
procedure AkcioFeldolgozas(i, j)
local:      PR : predikátum változók halmaza
begin
  S := ∅
  for ∀(d_k, U_k) ∈ DU^C(i) elemre
    PR := ∅
    for ∀u_kl ∈ U_k változóra
      if u_kl = ret(g) valamely g függvényre then
        TR ≫ tr
        if tr ≠ FB(g) then SzinkronizaciosHiba
        else FuggvenyFeldolgozas(g) endif
      endif
      u'_kl := CimFeloldas(u_kl)
      if u'_kl predikátum változó then
        PR ← u'_kl
      else
        U'_k ← u'_kl
      endif
    endfor
    U'_k ← p, amelyre LD(p) = max{LD(r)|r ∈ PR}
    d'_k := CimFeloldas(d_k)
    DynDep(d'_k) := ⋃_{u'∈U'_k} (DynDep(u') ∪ {LS(u')})
    LS(d'_k) := i
    if d'_k predikátum változó then
      LD(d'_k) := j
    endif
    S := S ∪ DynDep(d'_k)
  endfor
  S kimenetre, mint (x, i^j, V_i) kritérium dinamikus szelete
end
```

7.7. ábra. A globális algoritmus C-re (akció-feldolgozás)

```
program  IgenyvezereltAlgoritmusCre(P, C)
input:   P : program
         C = (x, i^j, V) : dinamikus szeletelési kritérium (feltesszük, hogy
         V = DU^C(i) használati halmazainak uniója)
output:  S : P dinamikus szelete C-re

global:  EHT : végrehajtási történet tábla
         AHT : cím-történet tábla
         tr : végrehajtási nyom eleme
         sc : láthatósági tartomány verem
local:   PR : predikátum változók halmaza

begin
  TR végrehajtási nyom rögzítése i^j-vel bezárólag
  TablakKiszamitasa
  S := ∅
  worklist ← i^j
  while worklist ≠ ∅
    k^l := legnagyobb l-lel rendelkező elem kivétele worklist-ből
    S ← k
    for ∀(d_m, U_m) ∈ DU^C(k) elemre
      PR := ∅
      for ∀u_mn ∈ U_m változóra
        if u_mn típusa skaláris vagy dereferencia then
          u'_mn := AHT^<(u_mn, l)
        else
          u'_mn := u_mn
        endif
        if u'_mn predikátum változó then
          PR ← EHT^<(u'_mn, l)
        else
          worklist ← EHT^<(u'_mn, l)
        endif
      endfor
      worklist ← p^r ∈ PR, a legnagyobb r-rel rendelkező akció
    endfor
  endwhile
  S kimenetre
end
```

7.8. ábra. Az igényvezérelt algoritmus C-re (főprogram)

```
procedure TablakKiszamitasa
begin
  (* ugyanaz mint a globális algoritmus főprogramja TR rögzítése után *)
end
procedure FuggvenyFeldolgozas(f)
begin
  (* ugyanaz mint a globális algoritmusnál *)
end
procedure CimFeloldas(x)
begin
  (* ugyanaz mint a globális algoritmusnál *)
end
procedure AkcioFeldolgozas(i, j)
begin
  for ∀(d_k, U_k) ∈ DU^C(i) elemre
    for ∀u_kl ∈ U_k változóra
      if u_kl = ret(g) valamely g függvényre then
        TR ≫ tr
        if tr ≠ FB(g) then SzinkronizaciosHiba
        else FuggvenyFeldolgozas(g) endif
      endif
      if u_kl típusa skaláris vagy dereferencia then
        u'_kl := CimFeloldas(u_kl)
        if AHT u_kl sorában az utolsó cím ≠ u'_kl then
          (j, u'_kl) hozzáadása a sorhoz
        endif
      endif
    endfor
    d'_k := CimFeloldas(d_k)
    i^j hozzáadása az EHT d'_k sorához
  endfor
end
```

7.9. ábra. Az igényvezérelt algoritmus C-re (akció-feldolgozás)

8. fejezet

Szeletelési algoritmusok mérései

„640 kilobájt legyen elég mindenkinek.”
— Bill Gates (1981-ben)

Az előző fejezetekben bemutatott dinamikus szeletelési algoritmusainkhoz prototípus implementációkat is készítettünk, amelyekkel számos mérést végeztünk el annak érdekében, hogy a módszerek számítási és tárfoglalási hatékonyságát vizsgáljuk különböző szempontok szerint. Jelen fejezetben áttekintjük a mérési környezetünket és módszerünket, megadjuk a teszt programok jellemző méreteit, továbbá némi adatot szolgáltatunk a kiszámolt szeletek méreteire vonatkozóan, és végül megadjuk az algoritmusok komplexitás méréseinek eredményeit.

Az algoritmusokkal korábban is végeztünk valamilyen szintű méréseket (ld. [8, 12, 46] cikkeket), de a jelen fejezetben sokkal részletesebb eredményeket mutatunk be.

8.1. Teszt-programok és tesztesetek

Kísérleteinkhez öt kis–közepes méretű szabadon elérhető, közismert C programot használtunk. Ezek a *bcdd* (BCD számok kezelése), az *unzoo* (*zoo* tömörített fájlok kitömörítője), a *bzip* (tömörítő program), a *bc* (tudományos számológép) és a *less* (sokoldalú szövegmegjelenítő). Az alábbi táblázat néhány adatot szolgáltat a programok méreteiről:

8.1. táblázat. A teszt-programok

program	fájlok	bájtok	sorok	utasítások	függvények
bcdd	5	16 163	442	78	5
unzoo	1	118 914	2 900	932	30
bzip	1	130 458	4 495	2 270	73
bc	20	312 721	11 554	3 441	138
less	43	639 035	21 488	5 373	363

Az első három oszlop a programhoz tartozó forrásfájlok számát, a forrás teljes méretét bájtban és az összes sorok számát adja meg (beleértve az üreseket is). A negyedik oszlop a statikus analízis során beazonosított különálló végrehajtható utasítások száma (ezt az algoritmusoknál I-vel jelöltük), és végül az utolsó oszlopban található a programot képező függvények száma.

8.1.1. A D/U programreprezentáció méretei

A statikus analízis egyik kimenete a D/U programreprezentáció, pontosabban a DU^C struktúra. A struktúra sorainak számát a program mérete határozza meg, ugyanakkor a teljes méretet a különböző változók száma és az egyes utasításokban felhasznált függőségek (használati változók) száma szabja meg. A változók lehetnek valódi és virtuális változók, ahogy azt az előző fejezetben részletesen bemutattuk. Az algoritmusok komplexitása szempontjából a változók száma, típus szerinti aránya fontos lehet az algoritmusok egyes részeiben, ezért elvégeztünk néhány egyszerű statisztikát erre vonatkozóan, amit az alábbi táblázatban foglalunk össze:

8.2. táblázat. A változók statisztikái

program	összes változók	skalárisok %	pred. %	deref. %	egyéb %
bcdd	179	31	24	2	43
unzoo	1 896	26	34	5	35
bzip	4 184	25	30	5	40
bc	6 898	19	34	6	41
less	10 605	18	41	4	37

Mint az várható is volt, a változók száma a program méretének függvényében változik, mégpedig közel lineárisan (hozzávetőlegesen 2-szeres szorzóval az utasítások számához képest). A különböző típusú változók aránya ugyanakkor viszonylag állandó értékeket mutat, a meghatározó változók gyakorlatilag a skalárisok és predikátumok (utóbbiakból van legtöbb). Meg lehet figyelni, hogy a skalárisok aránya kis mértékben csökkenni látszik a program méretével, míg a predikátumok csekély gyarapodást mutatnak. Dereferencia változóból általában viszonylag kevés van, ami azt is maga után fogja vonni, hogy végrehajtáskor a különböző memória címek számát legnagyobb mértékben a skalárisok száma fogja meghatározni (erre vonatkozó méréseink eredményét ld. a 8.3. alfejezetben).

A DU^C méretét a sorok számán kívül meghatározzák még az egyes sorok méretei, legfőképpen a használati halmazok elemeinek száma. Az alábbi táblázatban összegeztük a halmazok méreteire vonatkozó sarokszámokat:

8.3. táblázat. A D/U programreprezentáció méretei

program	utasítások	min	max	átlag	szórás
bcdd	78	1	19	5,4	4,7
unzoo	932	0	148	8,9	11,9
bzip	2 270	0	131	8,1	13
bc	3 441	0	270	12,6	21,5
less	5 373	0	121	6,9	8,2

Észrevehetjük, hogy lényegi összefüggés nem tapasztalható a program mérete és a használati halmazok mérete között, talán enyhe növekedés helyenként. Továbbá, az átlagos halmazméret viszonylag kicsinek mondható, ami az algoritmusok műveletigényét kedvezően

befolyásolja. Sőt, a maximális halmaz-méretek is messze elmaradnak az összes lehetséges változó számától.

8.1.2. Tesztesetek és szeletelési kritériumok

Az algoritmusok viselkedése nagyban függ a tesztesettől, amelyhez tartozó végrehajtás dinamikus szeleteit számoljuk, valamint magától a dinamikus szeletelési kritériumtól is. Emiatt először néhány reprezentatív példa futást (tesztesetet) definiáltunk tipikus programbemenetekkel (ezeket kézzel határoztuk meg és futtattuk le, nem vettünk igénybe automatikus tesztelő eszközt). Továbbá, valamennyi tesztesethez meghatároztuk az összes értelmes dinamikus szeletelési kritériumot, majd ezek mindegyikével végeztük a méréseket. A kritériumokat automatikusan generáltuk úgy, hogy először a statikus analízis fázisában legeneráltuk a statikus szeletelési kritériumokat, amelyeket minden végrehajtható utasításhoz melyben skaláris változó van definiálva rendeltünk hozzá. Igaz, így a hagyományos kritérium definícióhoz képest nem az összes lehetségest kaptuk, hanem a gyakorlati szempontból érdekeseket. Az alábbi táblázatban összefoglaltuk a tesztesetek és a statikus szeletelési kritériumok számát, valamint a tesztesetekhez tartozó végrehajtások lépésszámainak szélsőértékeit:

8.4. táblázat. Tesztesetek és statikus kritériumok

program	tesztesetek	lépésszámok	statikus kritériumok
bcdd	5	623 – 623	55
unzoo	13	7 469 – 1 449 852	493
bzip	18	561 – 48 691	1 040
bc	49	1 209 – 13 898	1 323
less	14	27 868 – 247 916	1 860

Ezután a fenti tesztesetek mindegyikére lefuttattuk a globális algoritmust, amely előállította az összes dinamikus szeletet, a szükséges statisztikákat a futásokról, valamint a dinamikus szeletelési kritériumok listáját (a statikus kritériumokat kiegészítve az érintett utasítások minden előfordulásával a végrehajtás során). Végül lefuttattuk az igényvezérelt algoritmust egyenként a kapott dinamikus kritériumokra, amely ugyancsak előállította a megfelelő dinamikus szeleteket és a futási statisztikákat. A két algoritmus által előállított szeleteket össze is tudtuk hasonlítani, amelyek azonosaknak bizonyultak.

Vegyük észre, hogy a lehetséges dinamikus kritériumok száma drasztikusan megnőhet, hiszen egyes utasítások sokszor végrehajtódhatnak egy-egy futás során. Ez meglehetősen sok futtatását igényelte volna az igényvezérelt algoritmusnak (a leghosszabb futásunkhoz 1,3 millió dinamikus kritérium tartozott). Sőt, az igényvezérelt algoritmus jelenlegi prototípus implementációjában nem használtunk semmiféle optimalizálást arra vonatkozóan, hogy hasonló futásokhoz a már kiszámított adatokat újra fel tudjuk használni. Ez azt jelenti, hogy minden futtatásra a végrehajtási történetet az elejétől fel kellett dolgozni, ami miatt egy teszteset minden kritériumára való futtatás hozzávetőlegesen négyzetes műveletigényű a végrehajtási történet hosszához képest. A fentiek miatt az igényvezérelt algoritmust programonként egy, véletlenszerűen kiválasztott teszteset összes kritériumával mértük.

8.2. Szeletek méretei

Mivel a globális szeletelési algoritmussal egyszerre számos dinamikus szeletet állítottunk elő, könnyen tudtunk a szeletek méreteire vonatkozóan is átfogó statisztikákat készíteni. Ebben az alfejezetben összegezzük ezen méréseinket.

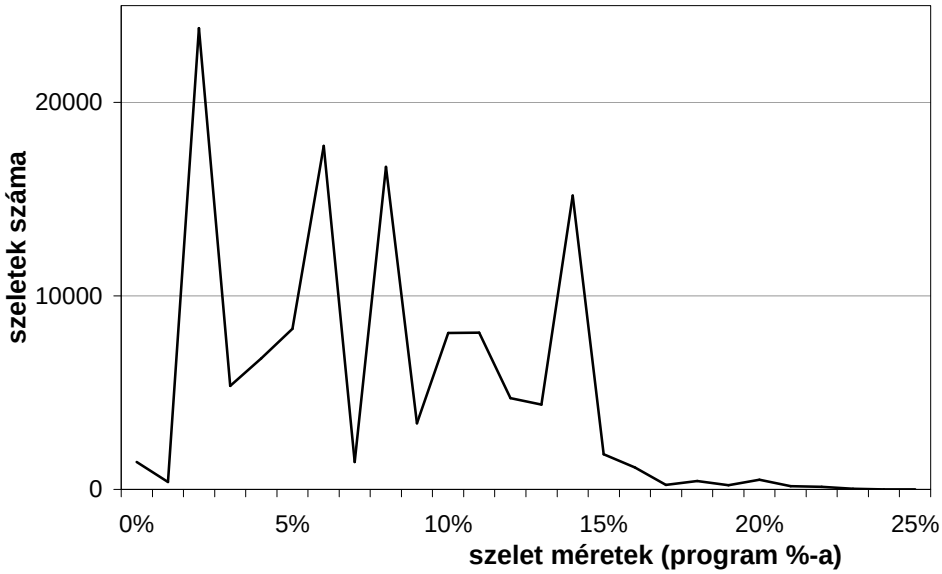
Minden tesztesethez megvizsgáltuk az összes dinamikus szelet közül a minimális, maximális és az átlagos méretet, majd összehasonlítottuk ezeket a program méretével és a tesztesethez tartozó végrehajtás hosszával is. Példaképpen, a legnagyobb programunk esetében (*less*) a következő értékeket mértük az egyes tesztesetekhez:

8.5. táblázat. A szeletek méretei a *less* programhoz

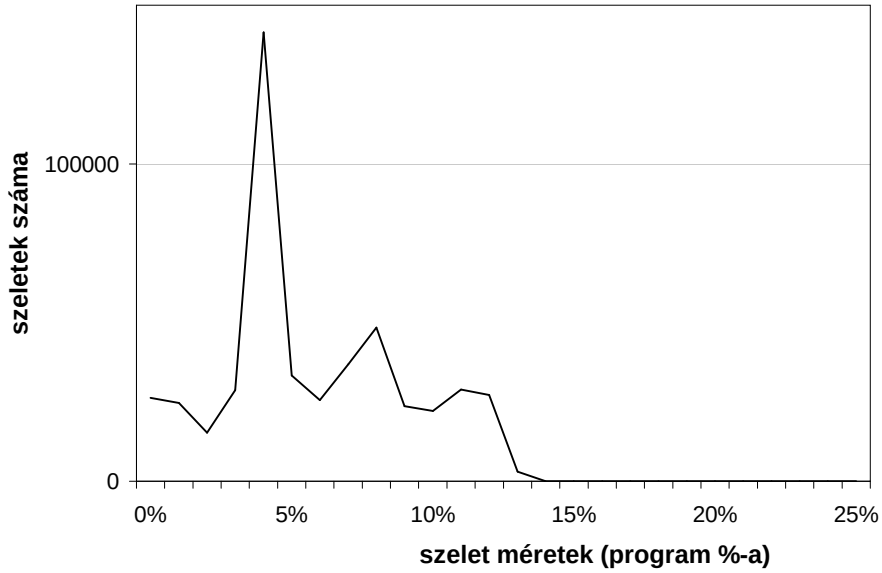
teszteset	lépések	min	max	átlag (%)
1	52 467	1	684	5,44
2	168 462	1	480	2,25
3	247 916	1	768	7,33
4	27 868	1	707	4,26
5	53 780	1	605	3,58
6	41 756	1	735	5,77
7	35 649	1	701	5,35
8	129 830	1	641	5,03
9	45 413	1	639	2,97
10	146 728	1	575	4,34
11	137 393	1	545	3,61
12	145 066	1	717	5,92
13	145 824	1	659	5,28
14	38 347	1	702	6,01

A méretek utasítás számban értendők, az átlagot a program méretéhez képest százalékban adtuk meg. Általában elmondható, hogy a szeletek méretei viszonylag állandó arányokat mutatnak a program méretéhez képest, a fenti programhoz például a szórás 113–297 között alakult. A másik fontos megfigyelés az, hogy a méretek nem függnek a végrehajtás hosszától. Ezen észrevételeink a többi program esetében is elmondhatók.

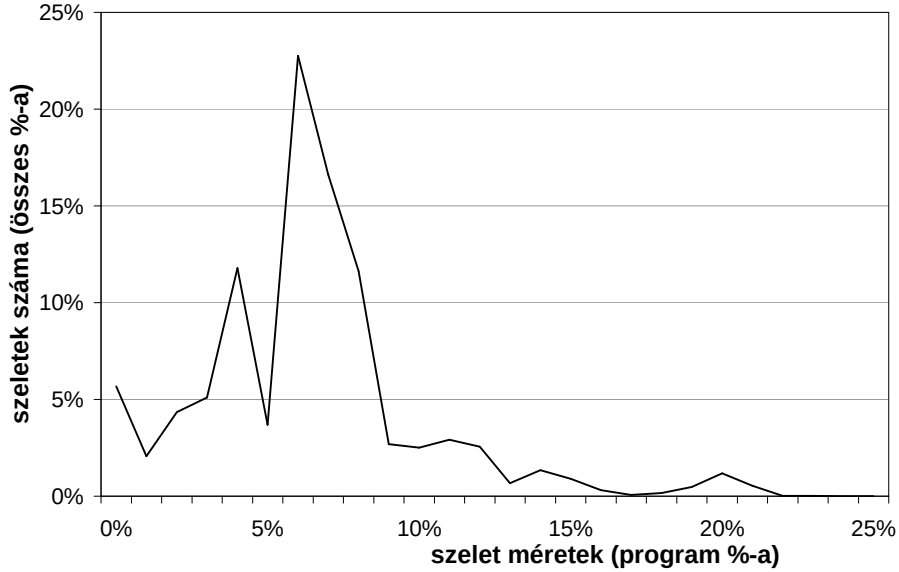
A szeletek méreteire vonatkozóan további adatokat szolgáltatunk a 8.1. ábrával. Itt hisztogram formájában mutatjuk be a kapott szeletek méreteinek eloszlását a programok méretéhez viszonyítva, az y tengelyen az adott méret-tartományba eső szeletek gyakoriságát feltüntetve. Példaképpen, az **(a)** diagram a *bzip*, a **(b)** pedig a *less* program hisztogramjait mutatja. A **(c)** diagram a teljes teszt anyag, azaz az összes program összes tesztesetéhez tartozó összes dinamikus szelet összesített adatait ábrázolja a szeletek összes számának százalékában (amely 2,6 millió szeletet jelent). Úgy tűnik, hogy a legtöbb szelet mérete a programok mindössze 4–9% százalékát éri el. A szeletek méreteit tovább vizsgáljuk a következő fejezetben az alkalmazások kapcsán.



(a) *bzip*



(b) *less*



(c) Összes program

8.1. ábra. Szeletek méretei

8.3. Számítási és tárterület komplexitás

Ezen alfejezetben bemutatjuk a két dinamikus szeletelési algoritmusunk hatékonyságát célzó méréseink eredményét. A műveletigény és tárterület hatékonyságot nem közvetlenül a futási sebesség és memóriafoglalás mérésével vizsgáltuk, hiszen a jelenlegi prototípus, optimalizálatlan implementációk nem képviselnek reális eredményt. Ehelyett az algoritmusok futása közbeni számítások különböző paramétereit mértük, amelyekkel az algoritmusok tárgyalásánál ismertetett elméleti komplexitási érveléseinket kíséreltük meg alátámasztani. Az implementációkba különböző helyekre speciális kiíró rutinokat építettünk bele, amelyek a belső adatszerkezetekre, lépésszámokra, stb. vonatkozó releváns információkat szolgáltatják. Az így kapott adatokat összegyűjtöttük az algoritmusok előző alfejezetekben ismertetett futtatásai során, majd ezeket utólagos feldolgozásoknak vetettük alá.

8.3.1. Használt memória címek száma

Mindkét algoritmus C programok analízisére való megvalósításának hatékonyságát jelentősen befolyásolja a futás során felhasznált különböző memória cellák száma, ugyanis azok működése a memória cellák közötti függőség-számításon alapszik. Ezért a futásokból összegyűjtöttük a címek számát, amelyeket az alábbi táblázatban összegzünk:

8.6. táblázat. Használt memória címek száma

program	skalárisok	deref.-k	min	max	átlag	szórás	din. eltérése
bcdd	55	4	34	34	34	0	-42%
unzoo	493	98	271	5 664	1 173	1 979	99%
bzip	1 040	194	82	1 586	985	499	-20%
bc	1 323	432	452	844	634	91	-64%
less	1 860	402	1 771	2 667	2 117	361	-6%

Az első két oszlop a skaláris és dereferencia változók (statikus) számát ismerteti (amelyekről már tudjuk, hogy többnyire a program méretével vannak arányban). Ezután találhatók a használt címek számának szélsőséges értékei, az átlag szám, valamint a szórás.

A kapott számok alapján összehasonlítottuk a memória hivatkozásokban érintett skalárisok és dereferenciák összegét az átlagos cím-számmal. Meglepő módon a dinamikus érték általában nagyon közel áll a statikus változószámhoz, többnyire némileg kisebb annál, amint azt a táblázat utolsó oszlopában láthatjuk (a statikus változószámhoz képest százalékban kifejezett eltérés). Kivételt képez a második program, amelynél körülbelül kétszer több dinamikus változót számláltunk össze, mint amennyi statikus változója van. Mind a negatív és pozitív eltérés is több dolog miatt adódhat: a tesztesetekkel nem fedtünk le minden utasítást, amelyben változót definiálunk, továbbá a mutatók értékei általában változhatnak a futás során, valamint a mutatók egybeesése (*alias*) is szerepet játszik. A két érték viszonylag szoros összefüggése mellet (0,73-as korrelációt mértünk) a különböző címek száma és az adott teszteset végrehajtási lépéseinek száma között is esetenként fellelhető némi összefüggés. Itt változó korrelációkat mértünk, a legkisebb 0,26 volt a *bzip* esetén, a legnagyobbat pedig a *less* program mutatta 0,68 értékkel. Az is megfigyelhető, hogy az említett dina-

mikus változószám eltérést legfőképpen a skalárisok száma határozza meg, hiszen azokból több van, mint dereferenciákból.

Következtetésként elmondhatjuk, hogy a szeletelési algoritmusok C nyelvre történő megvalósítása a címek használata szempontjából nem jelent kezelhetetlen többletköltséget.

8.3.2. Globális algoritmus

Mint ahogy azt a 6.4. alfejezetben már kifejtettük, a globális szeletelési algoritmus komplexitását a végrehajtási lépések száma és a halmazműveletek mennyisége, valamint a műveletekben részt vevő halmazok méretei határozzák meg döntően (a valódi sebességet természetesen még egyéb tényezők is befolyásolják, úgy mint a lemezre írás költsége, ha a kiszámított szeletek mindegyikét el kell menteni).

Halmazműveletek

Ahhoz hogy a halmazokról valós képet kaphassunk, az algoritmus implementációjával különböző statisztikákat írattunk ki a halmazokról és műveleteikről minden egyes iterációban. Az eredményeket összesítve azt kaptuk, hogy mindegyik programhoz hasonló arányok jöttek ki, amelyek közül az alábbi táblázatban a *bzip* programhoz kapcsolódóakat ismertetjük:

8.7. táblázat. A halmazműveletek statisztikái a *bzip* programhoz

teszteset	J	muv	$hszam$	$hmer_{atl}$	$hmer_{max}$
1	8 767	51 695	973	113,9	434
2	30 752	322 603	1 185	178,1	540
3	48 691	306 544	1 512	218,7	497
4	34 284	211 589	1 375	156,7	501
5	27 785	338 826	1 098	122,3	458
6	1 357	11 043	525	35,6	97
7	7 291	37 000	1 488	55,9	280
8	9 112	45 152	1 568	74,3	315
9	14 830	68 810	1 661	97,2	324
10	12 736	61 560	1 670	104,3	328
11	8 253	41 378	1 544	67,5	307
12	648	9 573	274	32,9	89
13	11 405	52 773	1 785	91,5	336
14	8 222	39 208	1 724	57,6	302
15	4 267	20 886	646	51,1	130
16	17 267	77 729	1 865	129,0	350
17	10 195	47 637	1 764	82,4	328
18	561	7 402	252	31,9	88

A különböző tesztesetekhez feltüntettük a végrehajtási lépésszámot (J), valamint a halmazokra vonatkozó különféle adatokat. A második oszlop az adott futáshoz tartozó összes halmaz-uniózás műveletek számát, a harmadik pedig az összesen kezelt $DynDep$ halmaz számát jelenti (ami nem más, mint a különböző változókra kiszámított szeletek száma a változók különböző előfordulásait nem számolva, nyilván ennél több szelet kerül kiírásra).

A következő két oszlopban a halmazműveletekben szereplő összes halmaz átlagos ($hmer_{atl}$) és maximális ($hmer_{max}$) méretét (elemszámát) adtuk meg.

Emlékezzünk vissza, hogy korábban az algoritmus legrosszabb esetének komplexitását $O(J \cdot V \cdot I \cdot \log(V \cdot I))$-ben állapítottuk meg. Most vizsgáljuk meg e képlet tényezőit a fenti értékek tükrében. Először is, $J \cdot V$ a halmazműveletek számát képviseli, hiszen minden lépésben legfeljebb V darab halmaz uniózására van lehetőség (a használati halmazok elemei révén). Méréseinknél ez konkrétan a muv érték. Ebből következik, hogy a muv/J értékekkel megkaphatjuk az egyes lépésekben elvégzett halmazműveletek átlagos számát, ami nem más, mint az adott lépésben feldolgozott használati halmazok összes elemszáma. A fenti programhoz ez az érték 4 és 15 közé esik, ami megfelel a fejezet elején ismertetett DU^C méréseinknek. Ugyanakkor, ez két nagyságrenddel kisebb, mint a legrosszabb esetnek megfelelő, és a képletben szereplő V érték (ami C esetében ugyebár, az összes használt memória címek száma).

Továbbá, a képletben szereplő I az utasítások száma, ami pedig a halmazműveletekben részt vevő halmazok legnagyobb méretét jelenti. Méréseinkben ezt a $hmer_{atl}$ értékek jelentik az adott teszteset átlagos méreteként (a maximális méret is vehető lenne, hiszen itt sem tapasztalható nagyságrendbeli különbség). Ez viszont gyakorlatilag nem más, mint a dinamikus szeletek átlagos mérete, hiszen minden kiszámított $DynDep$ halmaznak egy szeletet fogunk megfeleltetni. Ezen értékek a példánkban a program 1–10%-át teszik ki, ami megfelel a kiírt szeletek méreteire vonatkozó méréseinknek is.

Összegezve, a legrosszabb eset képletében szereplő tényezők körülbelül 4–5 nagyságrenddel (lineáris és logaritmikus tényezők együttvéve) vannak eltúlozva a méréseinkhez képest, valamint az átlagos esetre megadott képlet is alátámasztható az eredményeinkkel.

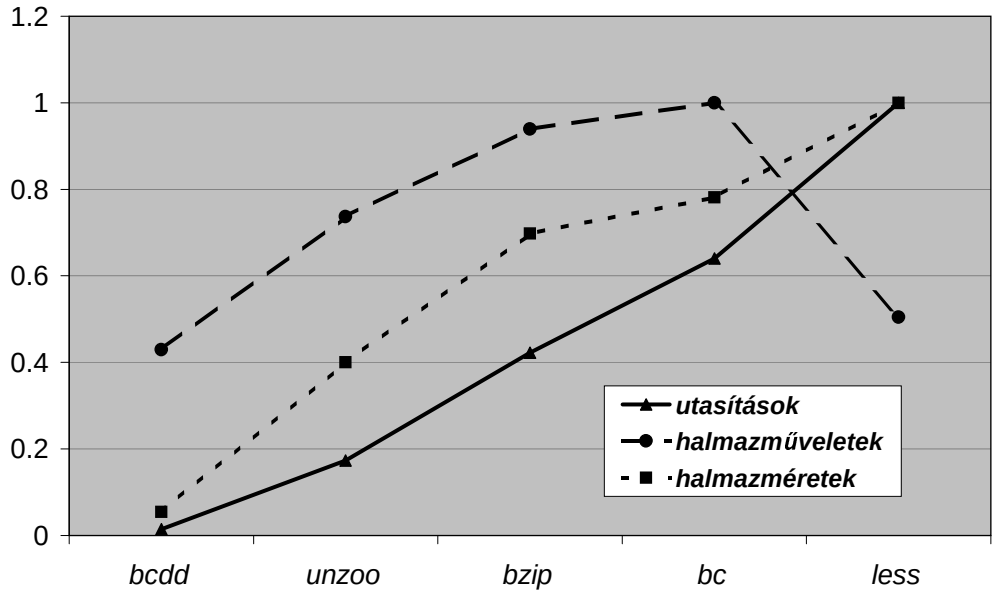
Tárigény

Ami a tárigényt illeti, a $DynDep$ halmazok tárolásának igényét a legrosszabb esetben a halmazok maximális számának és a maximális méretének szorzatával határoztuk meg, azaz $V \cdot I$. A halmazok maximális száma C esetében nem mondható meg, hiszen az a használt különböző memória cellák számával van meghatározva. A fenti táblázatban $hszam$-mal jelöltük az egyes tesztesetekhez tartozó halmazok számát. Ez magában foglalja az összes változóhoz tartozó $DynDep$ halmazok számát, tehát beleértve a virtuális változókat is, úgy mint függvényhívás-argumentumok (ezért ezekből több van, mint az alfejezet elején tárgyalt memória cellákból önmagukban). A fent példának használt *bzip* program esetében a maximális halmaz-szám tehát 1 865, az összes halmaz-méret maximuma pedig 540 (a program mérete 2 270). Ha a legrosszabb eset szorzatát összehasonlítjuk a $hszam \cdot hmer_{atl}$ értékekkel, akkor azt kapjuk, hogy átlagosan az utóbbiak körülbelül két nagyságrenddel kisebbek.

Halmazműveletek változása a programmérettel

Következő kísérletünkben megvizsgáltuk hogyan változnak az átlagos halmaz-méretek és halmazműveletek a program méretével. Fent ismertettük a *bzip* program eredményeit, de ugyanezeket a többi programra is elvégeztük. Nevezetesen, programonként minden tesztesethez kiszámítottuk az egyes lépésekben elvégzett halmazműveletek átlagos számát

(muv/J), és vettük ezek maximumát. Ez ugyebár, a legnagyobb átlagos használati halmaz elemszám, ami a fejezet elején ismertetett DU^C méréseink átlagához nagyon hasonló értékeket mutat. Azt kaptuk, hogy ez az érték az első négy program esetében növekedik a programmérettel 6,75–15,7 között, majd a legnagyobb programnál visszaesik 7,9-re, tehát gyakorlatilag nem vonhatunk le olyan következtetést, hogy ez a mérték egyértelműen a program-méret függvénye, és az is megtörténhet, hogy nagyobb program esetében kisebb lesz az átlagos halmazműveletek száma. Hasonlóan, meghatároztuk a maximális halmazméretek ($hmer_{max}$) maximumát is az összes tesztesetet figyelembe véve, amely esetben már monoton növekedést észleltünk, 42 és 774 között az öt programra, a programmérethez hasonlóan közel lineárisan némi meredekség-csökkenéssel. A 8.2. ábrán láthatjuk e két érték változását a programméret változásával összehasonlítva (az értékeket a megfelelő maximumokhoz normalizáltuk).



8.2. ábra. A globális algoritmus komplexitási tényezői a programméret mellett

Halmazműveletek változása a végrehajtási lépéssel

A globális algoritmus utolsó kísérletében megvizsgáltuk, vajon jelentős mértékben változnak-e a halmazméretek az algoritmus előrehaladtával a végrehajtási történet feldolgozásában. Ennek érdekében a végrehajtás minden egyes lépésében rögzítettük a halmazműveletekben szereplő halmazok maximális méretét, valamint az adott lépésben használt halmazok összesített elemszámát. A 8.3. ábrán láthatjuk ezen értékek alakulását a leghosszabb tesztesetünkre, valamint két program összesített eredményét (a halmazok aktuális méreteit adjuk meg a teljes lépésszám százalékosan elért részében). Az **(a)** diagram az *unzoo* program egyik tesztesetét mutatja, amely egy 240 kilobájtos szövegfájl kitömörítését végzi 1,45 millió lépéssel. Sem a maximális, sem az összes elemszám tekintetében nem tapasztalhatunk változást a végrehajtási történet előrehaladtával. A **(b)** és **(c)** diagramok mutatják a halmazméretek összesített változását a *bcdd* és *less* programok esetében (a legkisebb és a legnagyobb program). Ezen görbéket úgy kaptuk meg, hogy a tesztesetekhez tartozó egyenkénti görbék x tengelyeit a hozzátartozó teljes lépésszám százalékában határoztuk meg, majd az adott százalékértékhez tartozó halmazméretek átlagát tüntettük fel az y tengelyen. A **(c)** esetben

enyhe növekedést észlelhetünk mindkét mérőszám tekintetében, de azt is láthatjuk, hogy ez a növekedés lassuló tendenciát mutat.

Összegzés

A globális algoritmusunkra vonatkozó méréseink eredményeit összegezve elmondhatjuk, hogy a komplexitás megfelel az elméleti megfontolásainknak, nevezetesen a legrosszabb eset komplexitásánál nagyságrendekkel kedvezőbb az átlagos eset művelet- és tárigénye. Ezen kívül, a változók száma – ami alapvetően meghatározza a hatékonyságot – C programok analízise esetén is kezelhető, hiszen a használt különböző memória címek száma főleg a program méretével van összefüggésben, mintsem a végrehajtás hosszával. Végül, a legjelentősebb tényezők, a halmazműveletek száma és a bennük szereplő halmazméretek is hozzávetőlegesen lineáris összefüggésben vannak a program méretével, és a végrehajtási lépések számával sem változnak jelentős mértékben. Ugyanezen tényezők a tárigényt is kedvezően befolyásolják. Természetesen az előállított dinamikus szeletek lemezen történő tárolásának már jelentős költsége lehet, hiszen az összes különböző szeletek száma drasztikusan megnőhet.

A valódi futási sebességet és memóriafoglalást természetesen további konstans tényezők, a C programok miatti többletköltség, valamint egyéb, pl. fájlműveletek is befolyásolják. A C miatti többleteket a 7. fejezetben már tárgyaltuk. Ezek legfontosabbika, a dinamikus változószám (különböző memória cellák száma, valamint az összes változószám, $hszam$) viszont a fentiek alapján legfőképpen a program méretével van meghatározva.

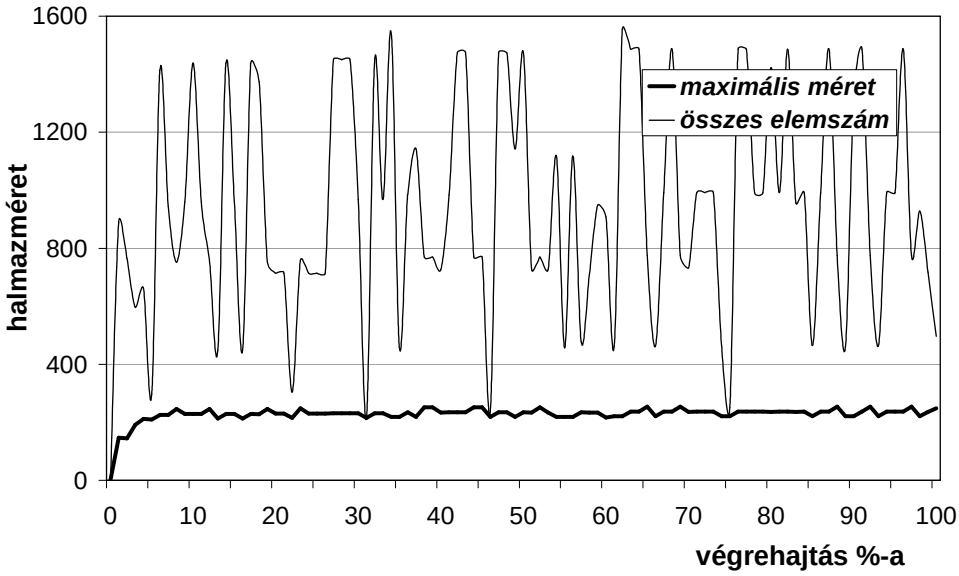
8.3.3. Igényvezérelt algoritmus

Az igényvezérelt dinamikus szeletelési algoritmus komplexitásával kapcsolatos méréseink a következő tényezők vizsgálatát jelentette (ld. a 6.5. alfejezetet): az EHT és AHT táblák méretei a rajtuk végzett műveletekkel, az algoritmus fő ciklusának iterációs száma, valamint a $worklist$ mérete és a dinamikus függőségek száma.

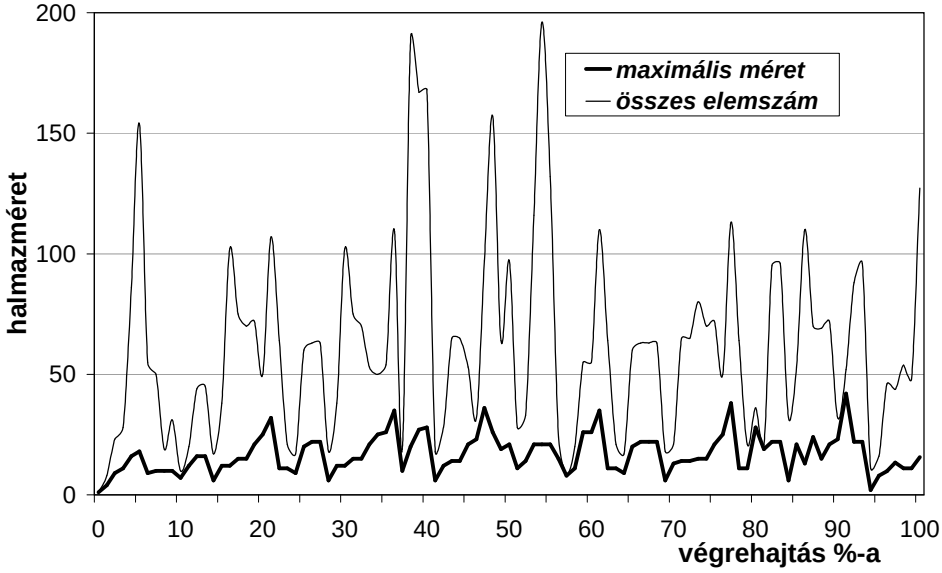
Mint ahogy azt a fejezet elején már kifejtettük, az igényvezérelt algoritmust nem tudtuk lefuttatni minden dinamikus szeletelési kritériumra (amelyeket a globális algoritmus állított elő). A jelen alfejezet méréseit ezért programonként egy teszteset összes dinamikus szeletelési kritériumára való futtatás adataival végeztük. Ez a következő számú futtatását jelentette az igényvezérelt algoritmusnak:

8.8. táblázat. Az igényvezérelt algoritmus futtatásai

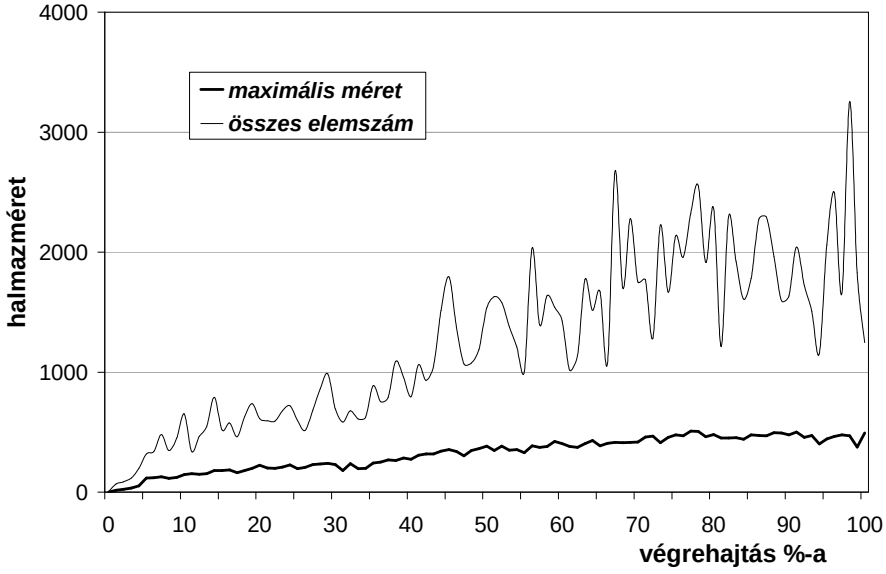
program	kiválasztott teszteset lépésszáma	dinamikus kritériumok
bcdd	623	459
unzoo	11 430	5 827
bzip	8 767	4 133
bc	10 672	4 847
less	38 347	11 519
összesen		**26 785**



(a) A leghosszabb teszteset



(b) *bcdd*



(c) *less*

8.3. ábra. A halmazméretek változása a végrehajtás során

Megjegyezzük, hogy a lépésszám a teszteset teljes lépésszámát jelenti, ennél kevesebb az alfejezet további részében tárgyalt végrehajtási lépésszám, ami a kritériumban szereplő akcióval bezárólag értendő.

Táblák méretei

Első kísérletünkben megvizsgáltuk az EHT és AHT táblák méreteit. Az EHT összes tárigénye azonos a lineáris végrehajtási történetével, kiegészítve a tároló struktúra többletköltségével. A tábla minden esetben annyi sort tartalmaz, ahány különböző változó lett definiálva a szeletelési kritérium lépésével bezárólag (ez legfeljebb V, a program összes változóinak száma, ami C esetében magában foglalja a használt memória címeket és az összes virtuális változót is). Utóbbira vonatkozóan az előző alfejezetben ismertettünk néhány eredményt. Az igényvezérelt algoritmust a változószám az EHT tábla elérésében befolyásolja ($EHT^{<}$ függvény komplexitása), ugyanis C esetében a sor megtalálása logaritmikus időt vesz igénybe, mert asszociatív tárolóra van szükség. Az elérés másik tényezője a sorban való keresés, amelyet a sorok hossza határoz meg. Erről tudjuk, hogy a legrosszabb esetben a teljes végrehajtási történet, és ekkor a tábla ebből az egyetlen sorból áll. A gyakorlatban azonban sokkal kedvezőbb a helyzet: az összes kritériumot tekintve a leghosszabb sor is mindössze 4–15%-a volt a megfelelő teszteset lépésszámának (viszont ne feledjük, hogy ez az arány romlik hosszabb végrehajtások esetén, hiszen a változók száma nem változik). Feltéve, hogy az akciók többé-kevésbé egyenletesen vannak elosztva az EHT sorok között,[1] az átlagos sorhosszak ($lépésszám/sorok\ száma$) 8,2, 20,2, 9, 7,7 és 12,3 értékűek az öt program vizsgált teszteseteire (növekvő sorrendben). A kapott értékek alapján a sorban való keresés kis műveletigényű, hiszen a legrosszabb eset logaritmikus tényezőjét átlagos esetben tovább csökkenti $\log(V)$-vel. Ugyanakkor várható, hogy hosszabb végrehajtások esetén a sorok hossza is növekedni fog és J lesz a meghatározó tényező, hiszen korábban már láttuk, hogy a használt változószám inkább a program méretével van összefüggésben.

Korábban már láttuk, hogy az AHT táblának annyi sora van, ahány különböző skaláris és dereferencia változó kerül használatra a végrehajtás során. Mint ahogy azt a jelen fejezet elején láttuk, az összes változók száma (a felső korlát) a program méretével arányos, valamint a skalárisok és dereferenciák részaránya is állandó. A sor elérése konstans idejű, tehát a valódi komplexitási tényezőt az $AHT^{<}$ függvény végrehajtása (keresés soron belül) jelenti. Ezt pedig a sorok hossza határozza meg, ami a változók címeinek váltakozási gyakoriságától függ. Ezt kivizsgálandó, megmértük a leghosszabb sorok elemszámát, valamint a sorhosszak átlagait mindkét változótípusra külön-külön. Ezeket összehasonlítottuk a végrehajtási történetek hosszaival, hiszen a két mérték szoros, lineáris összefüggésben áll egymással (egy program kivételével 0,98 feletti korrelációkat mértünk). Azt kaptuk, hogy a sorhosszak általában néhány ezerszer kisebbek, a leghosszabb sorok is a végrehajtási történetek hosszainak 1.6–4%-át tették ki. A skalárisok esetében a konkrét sorhosszak átlaga 3,8, 1,7, 3,7, 5,4 és 9,8 volt, viszonylag kis szórásokkal (0,63, 0,46, 2,18, 3,1 és 4,7). A dereferenciák esetében az átlagok némileg nagyobbaknak bizonyultak, mint ahogy az várható is volt. Az elemek átlagos száma 48, 58, 10, 7 és 42 volt, 10, 9, 4, 4 és 12 szórás-értékek mellett. Ezen értékek alapján az AHT tábla műveletigénye is kedvezőnek mondható.

[1] Programonként 3–4 véletlenszerűen kiválasztott EHT tábla sorhosszainak kiszámítottuk a szórását. Ezek a következő átlag értékeket mutatták (az adott végrehajtás hosszának százalékában): 3,3, 2,0, 1,0, 0,36 és 0,54.

Komplexitás lineáris tényezői

A 6. fejezetben az igényvezérelt algoritmus műveletigényét $O(J \cdot V \cdot \log(J))$-ben határoztuk meg. A lineáris tényező pontosabb meghatározására kétféle becslést is tárgyaltunk. Az első szerint a J tényező az iterációk számának (IT) felső korlátjaként adódik, a V pedig azon akciók maximális száma, amelyekkel a $worklist$-et egy iterációban bővítjük, vagyis a használati halmazok maximális mérete. A második becslés szerint a $J \cdot V$ szorzatot a képletben helyettesíthetjük $|E(DAG)|$-gal, ami nem más, mint a $worklist$ bővítési probálkozásainak száma. Minden teszt-futásra meg is mértük e tényezők konkrét értékeit, és azt kaptuk, hogy IT és $|E(DAG)|$ valóban szoros összefüggésben állnak egymással (a korreláció mindegyik programnál 0,99-nél nagyobb volt). Ez nem meglepő, hiszen tudjuk, hogy az iterációs számot megszorozva az átlagos használati halmazmérettel (U^{AVG}) megkapjuk a gráf éleinek számát. A szorzó 4,5–6,8 értékeket vett fel, tehát ez az érték adja az iterációnként dinamikusan felfedezett függőségek átlagos számát. Ez meg is felel a használati halmazokkal kapcsolatos méréseinknek a fejezet elején.

U^{AVG} mért értékéből látható, hogy a legrosszabb eset képletének első megközelítésében a V tényező jelentős túlzás az átlagos esethez képest, a példáink esetében két nagyságrendnyi az eltérés. Továbbá, ebből azt a következtetést is levonhatjuk, hogy a legrosszabb esetben is, amikor $IT = J$, a gráf mérete csak hozzávetőlegesen ötszöröse a végrehajtási lépésszámnak. Ezt a szorzót meg is mértük a legnagyobb gráfoknál, és azt kaptuk, hogy nem éri el az említett értéket, hanem 1,3–4 körül mozog. Ez viszont azt is jelenti, hogy ezen esetekben az iterációs szám is viszonylag magas volt: a lépésszámnak 65%-át is elérhette. Az alábbi táblázatban láthatjuk az iterációs számra vonatkozó konkrét méréseink összegzését:

8.9. táblázat. Az iterációs szám statisztikái

program	J_{veg}	IT_{veg}	IT_{atl}
bcdd	622	255	204
unzoo	11 411	1 564	482
bzip	8 765	71	1 185
bc	10 669	5 318	2 454
less	38 341	6 944	3 764

Az első két oszlop az adott teszteset legnagyobb akciójával megadott kritérium futásához tartozó lépés- és iterációs szám. Látható, hogy a legnagyobb, kritériumban is szereplő akció nem feltétlenül az utolsó, valamint az is, hogy az iterációs szám nem feltétlenül itt a maximális. Végül, a harmadik oszlop az átlagos iterációs számot adja. Megnéztük, hogy az iterációs szám hogyan viszonyul a hozzá tartozó lépésszámhoz, és nagyon változó viszonyokat kaptunk. Egyes esetekben az iterációs szám a lépésszámmal összemérhető, viszonylag magas értéket mutat. Azt is kiderítettük, hogy az átlagokat tekintve az $IT < |E(DAG)| < J$ viszony legtöbbször teljesül, bár vannak kivételek, amikor a gráf nagyobb a végrehajtás hosszánál.

Az iterációs szám jellegzetes mértéke után tovább kutatva, megvizsgáltuk annak viszonyát a dinamikus szeletek méretével (emlékezzünk vissza, hogy az algoritmus legalább annyit kell hogy iteráljon, ahány utasításból a szelet állni fog). Azt kaptuk, hogy az iterációszám körülbelül 5–8-szor nagyobb a szeletek méreténél (lineáris összefüggés a meghatározó). Azt

is megnéztük, vajon a végrehajtás hossza, vagy a szelet mérete az, amelyik jobban összefügg az iterációs számmal. Az előbbihez szerényebb korreláció mérhető (0,34), míg a szelet mérettel 0,9 körüli (ez alól csak az egyik program volt kivétel). Ez jó, mert a szeletek méreteiről tudjuk, hogy általában nem a végrehajtás hosszával arányosak, hanem a programmérettel, annak is viszonylag kis hányadát teszik ki. A 8.4. ábráról leolvashatók a szelet-méretekkel összehasonlított konkrét korrelációs értékek, az átlagos szorzók és az utóbbihoz tartozó szórás.

A szeletméretek, az iterációs szám, valamint a DAG méretek magas korrelációja még egy érdekes dologra enged következtetni: könnyen kiszámítható a fenti eredmények alapján, hogy a gráf körülbelül 22–54-szerese lesz a szelet méretének (olykor a végrehajtás hosszánál is nagyobb). Szerencsére, gráfot nem építünk az igényvezérelt algoritmussal, szemben a hagyományos DDG alapú módszerrel, amelynek gráfja még ennél is nagyobb.

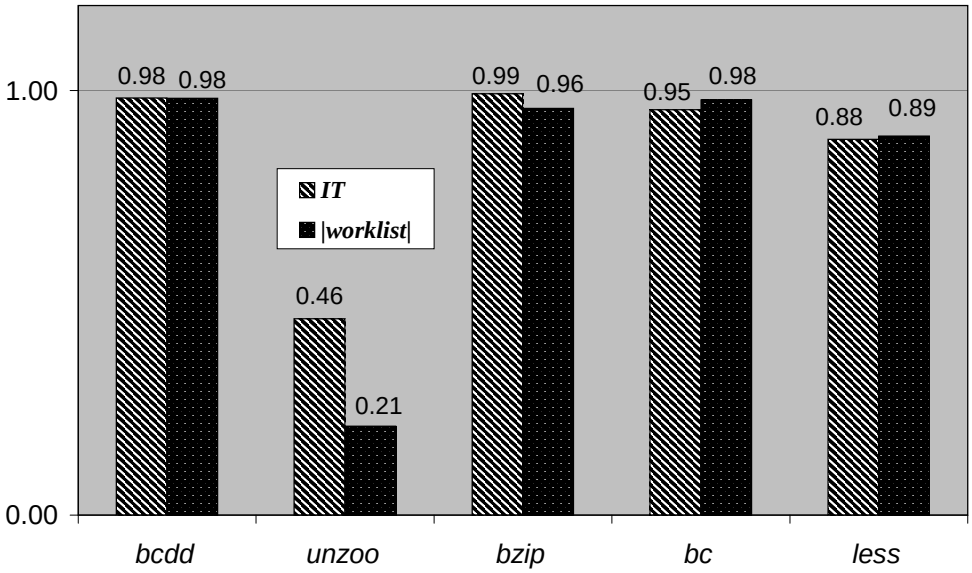
Komplexitás logaritmikus tényezői

Az igényvezérelt algoritmus komplexitásának $\log(J)$ tényezője adódik egyrészt az EHT táblában való keresés műveletigényéből, amit ezen alfejezet elején már tárgyaltunk, másrészt a $worklist$ egy elemmel történő bővítéséből. Utóbbi a $worklist$ aktuális méretétől függ, amely az iterációkkal folyamatosan változik, ezért erre vonatkozóan is részletes méréseket végeztünk. Tudjuk, hogy $|worklist| \leq IT \leq J$ a $worklist$ minden állapotában, így a maximális méretére is. Ezért vettük a maximális méreteit az algoritmus különböző futtatásaiban és ezeket összehasonlítottuk az iterációs számokkal. Meglehetősen magas korrelációkat mértünk az értékek között (0,92–0,99), ami annyira nem is meglepő, ha figyelembe vesszük, hogy egyrészt a $worklist$ általában tükrözi a DAG méretét (ld. az algoritmus tárgyalását a 6.5. alfejezetben), másrészt a fentiekből azt is tudjuk, hogy a gráf mérete az iterációs számmal is nagymértékben korrelál. Ezután megnéztük, hogy $|worklist|$ milyen mértékben kisebb IT-nél. Mind a konstans eltolást, mind a lineáris viszonyt megvizsgáltuk, és azt találtuk, hogy a lineáris arány átlagos értékei közelebb állnak egymáshoz, nevezetesen 17–49-szeres szorzókat találtunk (5–38 szórással). Másszóval, a $worklist$ mérete általában az iterációs szám 2–5,9%-a, és mivel az utóbbi is legtöbbször J-nél kisebb, a legrosszabb eset $\log(J)$ tényezője átlagos esetben tovább csökken. A konkrétan kimért $worklist$ méretek az alábbi táblázatban vannak összefoglalva (a szélsőértékek, az átlag és az adott teszteset legnagyobb akciójával megadott kritérium futásához tartozó értékek):

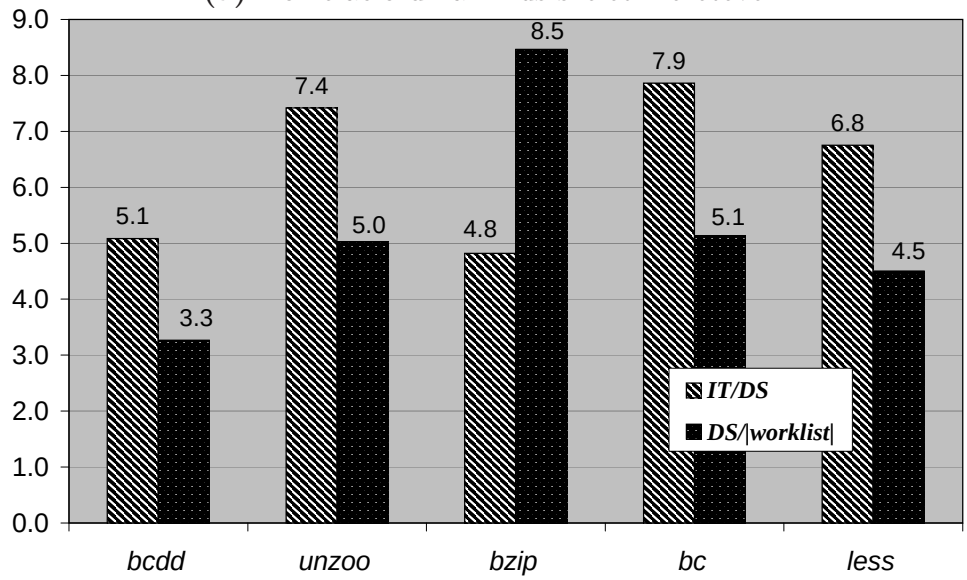
8.10. táblázat. A $worklist$ méretei

program	min	max	átlag	végső
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

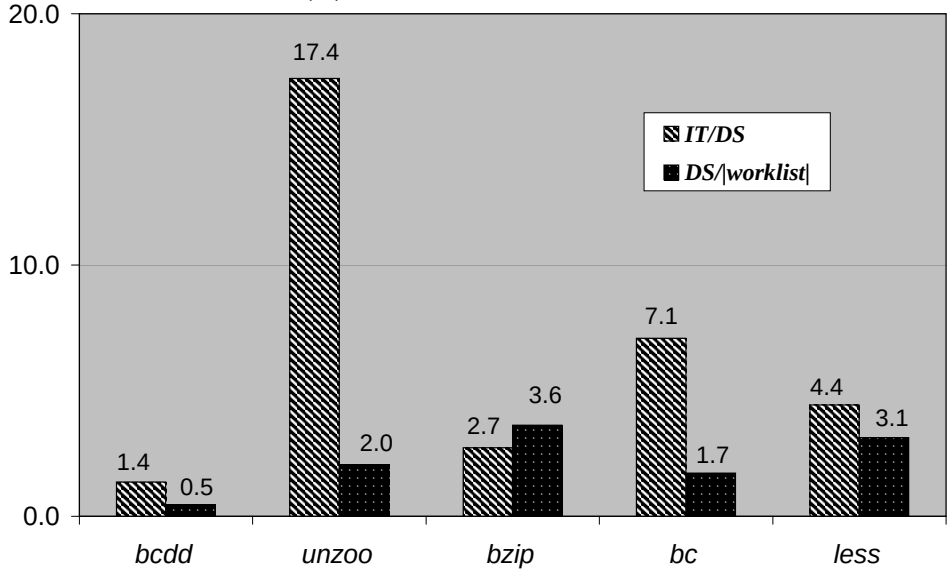
Hasonlóan az iterációs számhoz, a $worklist$ méretek is a szeletek méreteivel nagyobb korrelációt mutatnak, mint a végrehajtási lépésszámmal (a konkrét korrelációs értékek is



(a) Korreláció dinamikus szelet méretével



(b) Átlagos szorzótényező



(c) Átlagos szorzótényező szórása

8.4. ábra. A szeletek méretével való összehasonlítás

hasonlók, lévén az IT együtt mozog a $|worklist|$-tel). Olyan eredményre jutottunk, hogy a maximális $worklist$ méretek általában 3,3–8,5-ször kisebbek a szeletek méreteinél, ami a többi tényező fenti arányaival is összhangban van (ugyanakkor, az is előfordulhat, hogy a $worklist$ elemszáma nagyobb mint a szeleté). A 8.4. ábrán láthatók a konkrét korrelációs értékek, az átlagos szorzók és az utóbbihoz tartozó szórás.

Tárigény

A tárigényt alapvetően az EHT és AHT táblák méretei, valamint a $worklist$ maximális elemszáma határozza meg. Ezek mindegyikét a fentiekben tárgyaltuk, és az ismertetett eredmények alapján elmondhatjuk, hogy a legnagyobb költséget valószínűleg az EHT tábla képviseli, a többi tényező kezelhető. Mint tudjuk, elméletileg szükség van a teljes EHT tábla memóriában való tartására, de ez a probléma kiküszöbölhető az algoritmus bemutatásánál ismertetett módon.

Összegzés

Az igényvezérelt algoritmusról is elmondható, hogy a mérések alapján művelet- és tárigénye megfelel az elméleti megfontolásainknak, átlagos komplexitása lényegesen kedvezőbb a legrosszabb esetnek megadottnál. Nevezetesen, a fő tényezők – az iterációk száma és a $worklist$ mérete – jelentősebb mértékben vannak meghatározva a szeletek méretével, mint a végrehajtási lépésszámmal. A használati változók száma, ami lineáris tényezőt képvisel a komplexitásban, kezelhetőnek bizonyult, még akkor is, ha figyelembe vesszük, hogy a műveletigény C esetében kiegészül az $AHT^{<}$ függvény komplexitásával. Végül, a történettáblákról elmondhatjuk, hogy méréseink során azok többé-kevésbé egyenletes szerkezeteket mutattak, ezért a rajtuk végzett műveletek is arányosan megoszlanak (melyek logaritmikusan elvégezhetők). Természetesen a valódi futási sebességet és memóriafoglalást itt is további tényezők befolyásolják, úgy mint a globális algoritmus esetében. Valószínűleg a legfontosabb C miatti többletköltség az AHT tábla mellett a dinamikus változószám, amely az EHT tábla építését és elérését is befolyásolja, ez viszont az előzőekben ismertetett adatok alapján legfőképpen a program méretével van meghatározva.

8.4. Összegzés

A jelen fejezetben ismertetett mérési eredményeinkkel alátámasztottuk az elméleti komplexitási vizsgálódásainkat. Bebizonyosodott, hogy az igényvezérelt algoritmus általában gyorsabb, ha egy szelet kiszámítását nézzük, és ha nem tekintjük a táblák építését (vessük össze a lineáris tényezőkre kapott eredményeket, például a 8.9. táblázatban szereplő iterációs számokat). Ugyanakkor, a jövőben érdemes lesz a módszereket valós, ipari alkalmazásokban is kipróbálni, miután az algoritmusokhoz elkészítettük az optimalizált megvalósításokat, és azokat egy napi használatra is alkalmas eszközbe integráltuk.

A szeleteléssel kapcsolatos következő, utolsó fejezetünkben áttekintjük a módszereink meglévő és tervezett alkalmazásait.

9. fejezet

Dinamikus szeletelési algoritmusok alkalmazásai

„Köztudott, hogy a hibajavítás kétszer olyan nehéz mint megírni a programot. Szóval, ha a lehető legjobb formádat adod programírás közben, hogy fogod majd a hibákat kijavítani?"

— Brian Kernighan

Az 5. fejezetben áttekintettük a szeletelés legfontosabb alkalmazásait. Speciálisan, dinamikus szeletek használatát számos szoftverfejlesztési, -karbantartási és -megértési célra javasolja a szakirodalom. Ugyanakkor, rendkívül kevés utalás található arra vonatkozóan, hogy egy-egy szeletelési módszer konkrét megvalósítása valós környezetben lenne használva. A legtöbb dinamikus szeletelő implementáció prototípus szintjén marad (ugyanakkor statikus szeletelő eszközből több is fellelhető a piacon). A dinamikus szeletelési módszerek gyakorlati alkalmazásának valószínűleg egyik fő gátló tényezője az, hogy az ismert módszerek komplexitásuk miatt kevésbé alkalmasak valós méretű szoftverek analízisére. Az előző fejezetekben bemutatott algoritmusoknak is jelenleg prototípus implementációja létezik csak, de az ismertetett komplexitási vizsgálatok és mérési eredmények alapján valós szoftverek analízisére is alkalmasak lehetnek.

Jelen fejezetben áttekintjük azon alkalmazásokat, amelyekben a módszereinket már kipróbáltuk, illetve jelenleg is folyik a kutatás. Továbbá megemlítjük néhány további lehetséges felhasználásukat, ahol a korábbi módszerek alkalmazása nem lenne gazdaságos.

9.1. Nyomkövetés

A dinamikus szeletelés minden bizonnyal legfontosabb alkalmazása a nyomkövetésnél (*debugging*) van. A szeletelés első megfogalmazásában is ezt a célt jelölte meg Weiser [103]. Viszont ő statikus szeleteléssel foglalkozott, ugyanakkor nyomkövetéskor mindig egy konkrét futás függőségeire vagyunk kíváncsiak, ezért a dinamikus szeletelés még inkább alkalmas ezen szoftverfejlesztési tevékenység segítésére. Mitöbb, dinamikus szeletelést a kezdeti időkben csakis a nyomkövetésben alkalmaztak (ld. például Agrawal *és mások* [2], valamint Korel és Rilling [68] munkáit). Utóbbi szerzők abból a megállapításból indultak ki, hogy a dinamikus szeletelő módszerek kimenete a programnak egy szöveges formában prezentált részhalmaza, amelyet a programozónak tovább fel kell dolgoznia annak érdekében, hogy megtalálja a keresett, hibás viselkedésért felelős részeket. Ugyanakkor, hatékony nyomkövetéshez a program

minél könnyebb megértése elengedhetetlen, ezért ők néhány konkrét bővítését is javasolják a nyomkövető eszközöknek, amelyek révén hatékonyabb lehet e tevékenység. Ilyenek például az ún. befolyásoló változók megjelenítése (azok, amelyek kihatással voltak a kritériumnál szereplő változók értékeire), vagy a közreműködő utasítások (a végrehajtott utasítások közül azok, amelyek a szeletben is benne vannak). Továbbá, sokszor nincs is szükség a teljes szelet megjelenítésére, hanem csak a közvetlen függőségek követésére. Ez történhet például úgy, hogy kiindulunk a kritériumból és hátrafelé lépegetve követjük a függőségeket, amíg meg nem találjuk a hibás utasítás(oka)t. Általában nyomkövetési alkalmazásokban a dinamikus szelettől nem követeljük meg azt a tulajdonságot, hogy végrehajtható legyen.

Ezen nagyon egyszerű eszközökkel valóban hasznos szolgáltatásokkal lehet felruházni a nyomkövetőt, feltéve hogy rendelkezésünkre áll egy hatékony szeletelő algoritmus, amelyet interaktív üzemmódban is lehet használni. Mindkét általunk kidolgozott algoritmus alkalmas ilyen használatra is. E sorok írásakor fejlesztés alatt van a nyílt forráskódú GCC – GDB fordítóprogram és nyomkövető rendszer kiegészítése e funkcióval. E fejlesztésnek egyik nagyon fontos célja az, hogy kísérleti úton meglássuk, a két módszer közül melyik alkalmasabb inkább a nyomkövetésre. Elméletileg mindkettő alkalmazható, hiszen ilyen esetben a töréspontnál megállított programvégrehajtásra egy kritérium szerinti szeletet kell kiszámolni. Szemben a különálló szeletelő megvalósítással, amit az előző fejezetekben részletesen ismertettünk, itt az online megvalósítási módot kell követni (ld. a 7. fejezetet), ahol néhány dolgot a végrehajtási (nyomkövető) környezet szolgáltat, úgy mint a végrehajtásról szóló különböző információk.

Ha a globális algoritmust alkalmazzuk, a függőségek (és a szelet) a töréspontot elérve már mind meg vannak határozva, hiszen a végrehajtási nyomot párhuzamosan is feldolgozhatjuk (ez Zhang *és mások* terminológiáját használva teljes előfeldolgozást jelent [105, 107]). Itt a végrehajtásnál jelentkezik némi lassulás és többlet memória foglalás, viszont maga a szelet megjelenítése és az egyéb szolgáltatások végrehajtása azonnal adódik, sőt, ezek tetszőleges kritériumra elérhetők az algoritmus globális voltának köszönve. Ezzel szemben, az igényvezérelt algoritmus ilyen alkalmazása megfelel az előfeldolgozás mentes módnak, amelynek előnye, hogy végrehajtáskor mindössze a szükséges segéd-táblákat kell felépíteni. Majd a szelet meghatározása a töréspont elérése után történik, ami viszont esetenként lassú lehet, lévén hogy egy interaktív alkalmazásról van szó, ahol a gyors válaszidő kritikus. Továbbá, az igényvezérelt algoritmusnál minden egyes kritériumra újból le kell futtatni a szeletelő algoritmust. Természetesen számos lehetőség kínálkozik a módszerek továbbfejlesztésére, így például a hivatkozott cikkben szereplő, ún. korlátozott előfeldolgozás alkalmazása, amellyel az igényvezérelt szeletelést segíthetjük elő adott időközönként eltárolt, különböző összesített információkkal. A globális algoritmus hosszú előfeldolgozási igényét pedig a végrehajtási nyom hatékonyabb feldolgozásával tudjuk csökkenteni, például az ismétlődő részek tömörített tárolásával és összesített információkkal. Ezen továbbfejlesztéseken jelenleg is dolgozunk.

9.1.1. Releváns szeletek

Egyes alkalmazásokban, beleértve a nyomkövetést is, a hagyományos dinamikus szeleteknek van egy hátránya, amely minden dinamikus szeletelő algoritmust érint, beleértve Agrawal és Horgan [3], valamint Korel és Laski [66] algoritmusait, továbbá a jelen dolgozatban ismerte-

tett módszereinket is. A probléma ott jelentkezik, hogy a dinamikus szelet egyes esetekben nem foglal magában olyan utasításokat, amelyek ugyan nem befolyásolták a kritérium változóit, de más kiértékelés mellet már befolyásolták volna azokat (azt mondjuk, hogy a változók *potenciálisan* függnek az adott utasításoktól). Látható, hogy itt nem a dinamikus szeletelő algoritmusokkal van a gond, az előállított szeletek továbbra is pontosak olyan értelemben, hogy csakis a megvalósult függőségeket foglalják magukban. Nyomkövetéskor ugyanakkor fontos információ az, hogy egy utasítás megváltoztatása esetén az eredmény is megváltozott volna (a megváltozott utasítás lehet éppenséggel a hibáért felelős utasítás). A „volna" szócska arra utal, hogy itt már némi statikus függőségi információval is számolni kell, ugyanis az utasítások összes lehetséges kiértékelésének figyelembe vétele már statikus feldolgozás.

A probléma lényegét a 9.1. ábrán szereplő példán keresztül könnyen illusztrálhatjuk. Az **(a)** ábrán láthatunk egy példaprogramot, amelynek egyik dinamikus szeletét a **(b)** ábrán besatírozott utasítások adják. Láthatjuk, hogy a szelet nem tartalmazza a 2., 5. és 10. utasításokat. Ez helyes, hiszen az `s` változó értékének meghatározásában tényleg nem játszottak szerepet. Másszóval, ha ezen utasításokat (és a 11.-et) elhagynánk, ugyanazt az eredményt kapnánk a kritériumnál. Ugyanakkor, nyomkövetési szempontból ez nem kielégítő, mert ha a hibás utasítás az 5. (pl. `a += 2;` a helyes), akkor `s` értéke megváltozik a kritériumnál, és ezt nyomkövetésnél jó lenne észrevenni.

```
    #include <stdio.h>
    int n,a,b,i,s,x;
    void main()
    {
 1.   scanf("%d", &n);
 2.   scanf("%d", &a);
 3.   x = 1;
 4.   b = a + x;
 5.   a += 1;
 6.   i = 1;
 7.   s = 0;
 8.   while (i <= n) {
 9.     if (b > 0)
10.       if (a > 1)
11.         x = 2;
12.     s += x;
13.     i++;
      }
14.   printf("%d", s);
    }
```

(a)

```
#include <stdio.h>
int n,a,b,i,s,x;
void main()
{
  scanf("%d", &n);
  scanf("%d", &a);
  x = 1;
  b = a + x;
  a += 1;
  i = 1;
  s = 0;
  while (i <= n) {
    if (b > 0)
      if (a > 1)
        x = 2;
    s += x;
    i++;
  }
  printf("%d", s);
}
```

(b)

```
#include <stdio.h>
int n,a,b,i,s,x;
void main()
{
  scanf("%d", &n);
  scanf("%d", &a);
  x = 1;
  b = a + x;
  a += 1;
  i = 1;
  s = 0;
  while (i <= n) {
    if (b > 0)
      if (a > 1)
        x = 2;
    s += x;
    i++;
  }
  printf("%d", s);
}
```

(c)

9.1. ábra. Példaprogram a dinamikus és releváns szeleteivel a $(\langle \mathtt{a} = 0, \mathtt{n} = 2\rangle, 14^{19}, \{\mathtt{s}\})$ kritériumhoz

Az ilyen jellegű potenciális függőségek kezelésére Agrawal *és mások* [4] bevezették az ún. *releváns szelet* fogalmát, amelyet inkrementális regressziós tesztelésnél alkalmaztak. Lényegében a releváns szelet nem más, mint a dinamikus függőségeken alapuló dinamikus szelet (DDG-módszer, vagy az általunk ismertetett algoritmusok), kiegészítve a potenciális vezérlési függőségekkel és azok adatfüggéseivel. A fenti példából is látszik, hogy a rele-

váns szeletek nyomkövetésnél is rendkívül hasznosak (a példa releváns szelete a (c) ábrán látható). Ilymódon a releváns szelet mindig a dinamikus szelet kibővítése, ahol a bővítés statikus információ alapján történik, konkrétan egy bővített program-függőségi gráf (PDG [54, 81]) felhasználásával.

A potenciális függőségek számításának lényege a függőségi gráf bővítésében van az ún. feltételes függőségekkel (erre egy példa a 10. és 12. utasítás között fellépő függőség az x változóra és 10. *igaz* kiértékelésére vonatkoztatva). A feltételes függőségek egy adott bemenetre történő dinamikus szelet meghatározásakor potenciális függőségeket fognak eredményezni, ha az adott feltételes predikátum kiértékelése ellentétes a gráfban szereplő értéktől.

A [46] publikációban megadtuk a globális algoritmusunk egy kiegészített változatát releváns szeletek meghatározására, ahol a potenciális függőségek számítására vonatkozó további részletek megtalálhatók. Természetesen a releváns szeletek számítása többletköltséggel jár, esetenként nem is elhanyagolhatóval, hiszen a program-függőségi gráf számítása jelentős számítási és tárterület komplexitással bír.

Az említett cikkben közzétettük a globális releváns szeletelő algoritmusunk prototípus implementációjával végzett kísérleteink eredményeit is. Ezek összhangban voltak a később végzett szeletméretekkel kapcsolatos részletesebb méréseinkkel a dinamikus és statikus szeletek viszonyának tekintetében (ld. 8. fejezetet és a következő alfejezetet). Azt kaptuk, hogy míg a statikus szeletek átlagosan a program 58%-át, a dinamikus szeletek pedig 5%-át teszik ki, addig a potenciális függőségek további 2%-kal növelik a dinamikus szeleteket, így a releváns szeletek átlagosan 7%-ot képviselnek.

9.2. Uniós szeletek használata a szoftverkarbantartásban

Az utóbbi időben a szeletelési módszerek alkalmazása a nyomkövetésen túl is egyre jelentősebb. Ennek egy példája a szoftverkarbantartás, ahol hagyományosan a statikus szeleteket használják különböző karbantartási és programmegértési feladatokhoz, ld. például [14, 41, 54, 89]. A szelet haszna általában a vizsgálandó program annak egy részére való csökkentésében van, miáltal az adott cél elérésének érdekében csak a kapott részprogramot kell tovább vizsgálni. Sajnos, sok esetben a statikus szeletek túlságosan nagyok ahhoz, hogy hasznos információval szolgáljanak a felhasználónak. Ezzel szemben, a dinamikus szeletek sokkal pontosabb információt képesek szolgáltatni, hiszen a szeletek lényegesen kisebbek (sajnos előttünk nem ismert másik tudományos publikáció, amely a dinamikus és statikus szeletek méreteinek viszonyát taglalná). A dinamikus szelet azért pontosabb, mert csak egy tesztesetet (programfutást) vizsgálva adja meg a függőségeket. Egyes alkalmazásokban pontosan ez a cél (mint pl. nyomkövetés), ugyanakkor más alkalmazásokban több teszteset (vagy az összes lehetséges) vizsgálatára van szükség. Természetesen az összes tesztesethez tartozó dinamikus szelet lefedése legtöbbször lehetetlen, továbbá egy költséges szeletelési algoritmus (mint pl. Agrawal DDG-alapú módszere) használata gyakorlatilag lehetetlenné teszi az ilyen jellegű analíziseket.

A fenti problémát – tehát azt hogy az analizálandó program vizsgált részét anélkül csökkentsük, hogy elveszítenénk a statikus szeletek általánosságát –, számos kutató vizsgálta

már. Az ún. feltételes szeletelés (conditioned slicing) során a program egy részhalmazát határozzuk meg, amely megtartja az eredeti program viselkedését egy adott (statikus) szeletelési kritériumra és a bemenetek lehetséges halmazának egy megszorítására [16, 51]. Ez a módszer alapvetően statikus, anélkül hogy tesztesetek ténylegesen végrehajtásra kerülnének. A módszer gyakorlati alkalmazhatósága megkérdőjelezhető a bonyolult számítási mechanizmusa miatt. Hibrid szeletelési módszerekkel többen is foglalkoztak, amikor is a statikus és dinamikus szeletelést valamilyen módon kombinálták. Például, Venkatesh [99] munkájában az ún. kvázi-statikus szeletelést alkalmazza, ahol egyes bemeneti változók értékeit rögzíti, így elérve dinamikus eredményt. Rilling *és mások* is egy hibrid szeletelési módszert ismertetnek, ahol a dinamikus szeletelést azáltal segítik elő, hogy először statikus szeleteléssel csökkentik az analizálandó program méretét [87].

Hall [47] olyan alkalmazásban használja fel a dinamikus szeleteket, ahol a különböző futásokhoz tartozó szeletek kombinálásával a programnak egy olyan végrehajtható részét állítja elő, amely például a program újratervezésénél vagy újrafelhasználásánál jelent segítséget. A módszer (melynek neve *szimultán dinamikus programszeletelés*) a felhasznált dinamikus szeletelési algoritmustól elvárja, hogy végrehajtható szeleteket állítson elő, továbbá maga az eljárás egy meglehetősen komplex iteratív algoritmuson alapul. Hall módszeréhez hasonlóan, mi is kísérleteztünk dinamikus szeletek kombinálásával, csak mivel a mi esetünkben nem volt követelmény a végrehajthatóság (a szoftverkarbantartás számos egyéb területén nem az), lényegesen egyszerűbb eljárást alkalmazhatunk. Nevezetesen, a dinamikus szeletek közönséges egyesítését (Hall arra a megállapításra jutott, hogy az egyszerű unió nem eredményez végrehajtható szeletet). Az alábbiakban ismertetjük a megközelítésünket és a mérései eredményeinket.

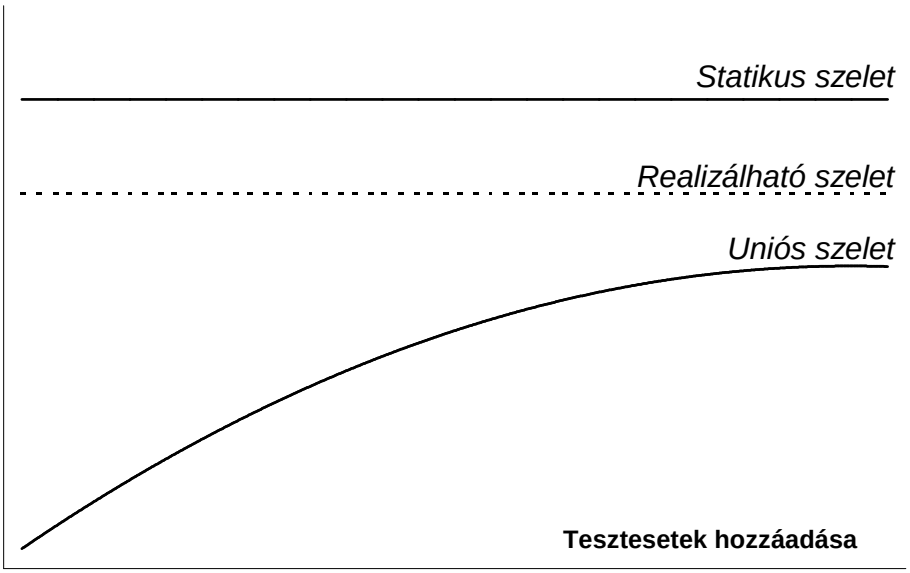
9.2.1. Realizálható és uniós szeletek

Abból indulunk ki, hogy tesztesetek egy jól megválasztott halmazával olyan dinamikus szeleteket állíthatunk elő, amelyek egyesítése majdnem olyan általánosan érvényes a többi lehetséges futásra nézve, mint a statikus szelet, viszont annál lényegesen kisebb. Ezen feltevésünket mérésekkel is igazoltuk, ezért járható az az út, hogy a program vizsgálata során először a kiválasztott tesztesetekhez tartozó programrészeket tekintjük, majd szükség esetén a maradékot, ezáltal csökkentve a karbantartás költségeit. Az uniós szeletekkel kapcsolatos munkánkat a [8] cikkben publikáltuk. Megjegyezzük, hogy Danicic *és mások* a [22] cikkükben a mi eredményeink által motiválva végrehajtható uniós szeletek számítására adtak meg egy módszert.

Adott utasításhoz tartozó, hátrafelé irányuló *uniós szeletnek* egyszerűen (egy konkrét dinamikus szeletelő algoritmussal előállított) dinamikus szeletek unióját értjük különböző tesztesetekre, de ugyanazon kritériumra (az utasítás utolsó előfordulásával). Elméletileg, ha az összes lehetséges futásra kiszámítanánk a dinamikus szeleteket, majd azokat egyesítenénk, akkor megkapnánk a programnak egy olyan részhalmazát, amely az összes megvalósítható függőséget magában hordozza. Ezért ezt a csak elméletben létező szeletet *realizálható szeletnek* fogjuk nevezni. Vessük össze e szeletet a *precíz* szelettel, ahol ugyancsak a függőségek minimális halmazát rögzítjük. A különbség az, hogy a precíz szelet a változók kiszámított értékeit veszi alapul, és nem egy konkrét, algoritmikusan kiszámítható szeletet. A precíz szelet kisebb lesz, mert nem tartalmaz olyan utasításokat, mint `x=x;`, lévén hogy az nem

módosítja a változó értékét. Ugyanakkor ennek csak elméleti jelentősége van, mert az ilyen összefüggések általánosan nem meghatározhatók. Ezzel szemben a realizálható szeletet gyakorlati eszközökkel közelíthetjük, és a közelítés pontossága csak a végrehajtott tesztesetek jóságától függ.

A statikus szeletet a realizálható szelet felső korlátjának tekinthetjük, hiszen minden realizált függőség valamely dinamikus szelet révén benne kell hogy legyen a statikus szeletben is (emellett a statikus szeletelő algoritmus kifinomultsága függvényében több-kevesebb „fölösleges" utasítás is benne lehet a szeletben, de a realizálható (és precíz) szeletet egyik algoritmus sem szolgáltathatja). Másfelől, minden uniós szelet alsó korlátja lesz a realizálható szeletnek (ld. a 9.2. ábrát). Minél több tesztesettel számítjuk ki az uniós szeletet, az annál jobban fogja közelíteni a realizálható szeletet. Méréseinkből arra a következtetésre jutottunk, hogy újabb tesztesetek hozzáadásával egy idő után az uniós szelet mérete lassabban fog növekedni, tehát egyre inkább megközelíti a realizálható szeletet. Legszerencsésebb az az eset, amikor a statikus és az uniós szelet egybeesik, ekkor ugyanis a realizálható szeletet is megtaláltuk, és további tesztesetek sem hoznának be újabb utasításokat. Sajnos ez nagyon ritkán következik be, hiszen a teszteseteknek egy nagyon széles skálája szükségeltetik hozzá. Sőt, egy teljes lefedettséget biztosító teszteset-halmaz (akár utasítás- akár vezérlési folyam szinten) sem biztosítja a realizálható szelet elérését. Legtöbb gyakorlati esetben az uniós szelet gyarapodása viszonylag gyorsan lecsökken adott számú, jól megválasztott reprezentatív teszteset után, amiből arra következtethetünk, hogy a realizálható szeletet is jól megközelítettük. Nagy gyakorlati jelentőségű, hogy kísérleteink révén azt tapasztaltuk, hogy az így kapott programrész még mindig lényegesen kisebb lesz a statikus szeletnél.



9.2. ábra. A realizálható szelet közelítése

9.2.2. Uniós szeletek számítása és kiértékelése

Az uniós szeleteket az (X, i, V), ún. uniós szeletelési kritériumra számíthatjuk ki, ahol $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ a P program bemeneteinek egy halmaza (a tesztesetek), továbbá i és V a statikus szeletelési kritérium szokásos paraméterei. Ezek után az uniós szelet nem

más, mint az $(\mathbf{x}_k, i^{j_k}, V)$ kritériumra kiszámított dinamikus szeletek uniója, ahol $\mathbf{x}_k \in X$, valamint az i^{j_k} akciók az i utasítás utolsó előfordulásai a megfelelő k-adik futásokban.

Vegyük észre, hogy uniós szeleteket bármely dinamikus szeletelési algoritmus alkalmazásával számíthatunk. Ugyanakkor, az algoritmusnak eléggé hatékonynak kell lennie, hiszen több szeletet kell meghatározni egy adott vizsgálathoz. A mi kísérleteinkben a globális szeletelési algoritmusunkat használtuk, és elmondhatjuk, hogy hatékonynak bizonyult a feladat elvégzésére. Azért választottuk a globális algoritmust, mert az egyidejűleg sok dinamikus szeletet állít elő egyetlen teszteset feldolgozásakor. Így, a módszer kiértékeléséhez könnyedén tudtunk sok adatot előállítani.

Kísérleteinkhez összeállítottunk egy mérési környezetet, amely a globális szeletelőből, az uniós szeleteket kiszámító eszközből, egy statikus szeletelőből, valamint egy megjelenítőből áll. A példaprogramokat és teszteseteket szkript tartalmazta, amely vezérelte a mérési folyamatot. Első lépésben a globális algoritmussal előállítottuk az adott tesztesetekhez a dinamikus szeleteket és azokat lemezen eltároltuk. Ezután, az uniózó eszközzel kiszámítottuk az uniós szeleteket, valamint még további statisztikákat, úgy mint a tesztesetek által nyújtott lefedettséget. Mindegyik uniós kritériumra meghatároztuk a megfelelő statikus szeleteket is egy külső statikus szeletelő eszközt alkalmazva. A kapott szeleteket és statisztikákat ezután kiértékeltük, amit az alábbiakban ismertetünk.

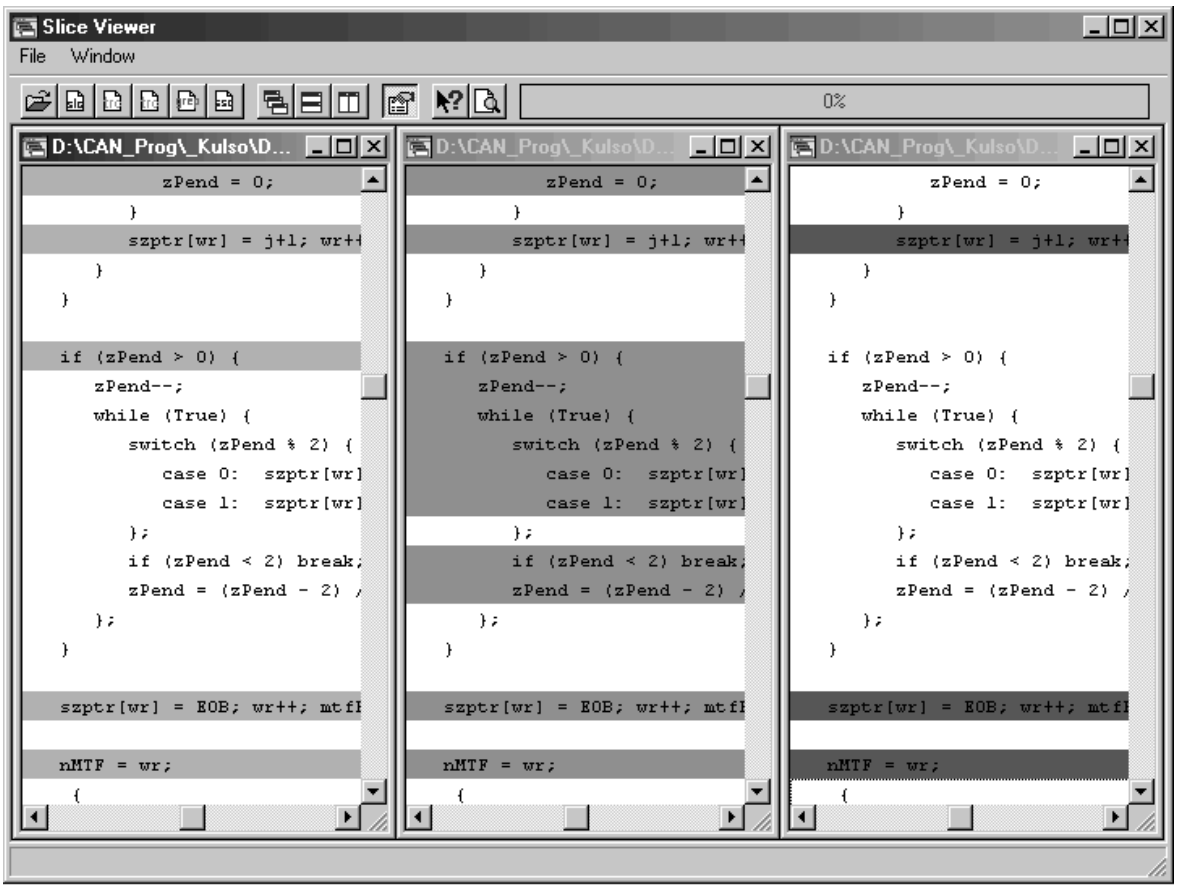
A mérési környezetünknek része még egy szelet-megjelenítő eszköz, amely a különböző szeleteket képes szinkronizálva megjeleníteni, azaz ugyanazon programsorokat egymás mellé helyezett szövegszerkesztő ablakokban láthatjuk, a szelethez tartozó sorokat különböző színűekre festve. Ezáltal a szeletek vizuális vizsgálata nagyon kényelmesen elvégezhető. A megjelenítő egyéb információ megjelenítését is el tudja végezni, úgy mint a külső eszközzel előállított statikus szeleteket, vagy a végrehajtási történetet. Az eszköz egy tipikus képernyőjét láthatjuk a 9.3. ábrán.

9.2.3. Mérések

Kísérleteinkkel azt akartuk megvizsgálni, hogy a kiszámolt uniós szeletek milyen mértékben közelíthetik meg a realizálható szeletet, vagyis újabb teszt esetek hozzáadásával hogyan alakul az uniós szelet mérete. Ezen kívül, arra is kíváncsiak voltunk, hogy a kapott dinamikus szeletek és az uniós szeletek méretei hogyan viszonyulnak a statikus szelet méreteihez. Tudomásunk szerint, eddig csak Venkatesh [100] foglalkozott a dinamikus szeletek méreteinek vizsgálatával, viszont ő nem hasonlította ezeket össze a statikus szeletek méreteivel, hanem különböző változó típusokhoz tartozó szeletekkel kísérletezett.

Méréseinkhez a 8. fejezetben már bemutatott példaprogramok közül a három legnagyobbat használtunk fel (*bzip*, *bc* és *less*), ugyanazon tesztesetekkel. A méretekre vonatkozó kiértékelést kézzel kiválasztott kritériumokra végeztük el (154, 57 és 50 darabra), így valós használati eseteket szimulálva. Ezen kritériumokat úgy választottuk meg, hogy minél szerteágazóbbak legyenek a hozzájuk tartozó szeletek függőségei, ezáltal az uniós szeletek minél nagyobb részét lefedjék a programoknak.

Első kísérletünkben megfigyeltük az uniós szeletek alakulását a teszt esetek hozzáadogatása során. Lépésenként meghatároztuk az uniós szeletek méreteit az egyes kritériumokra (utasításszámban), majd az így kapott méretsorozatok átlagát is kiszámítottuk. Az eredmény a 9.4. ábrán látható. A diagramokon csak néhány tipikus kritériumhoz tartozó gör-

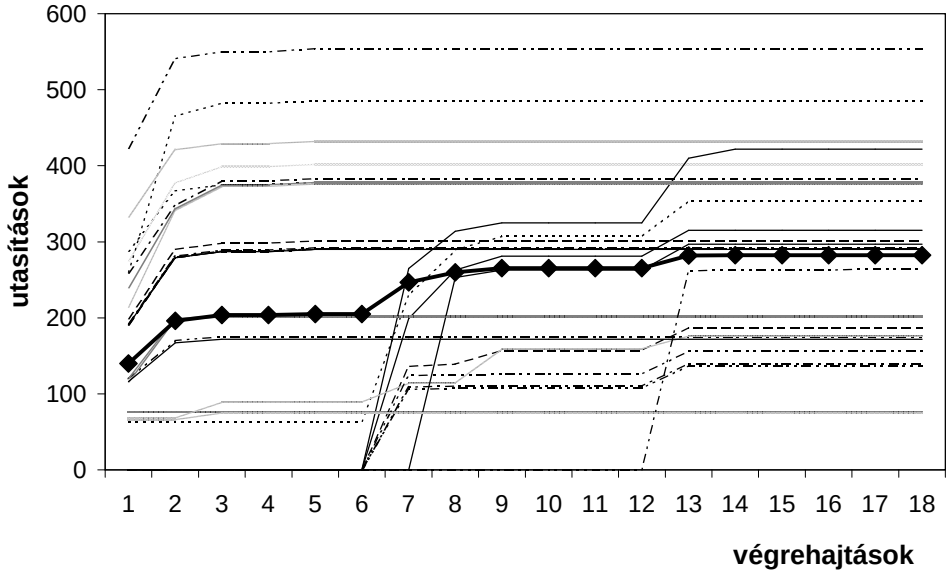


9.3. ábra. A szelet-megjelenítő egy tipikus képernyője (balról jobbra: végrehajtási nyom, statikus szelet, uniós szelet)

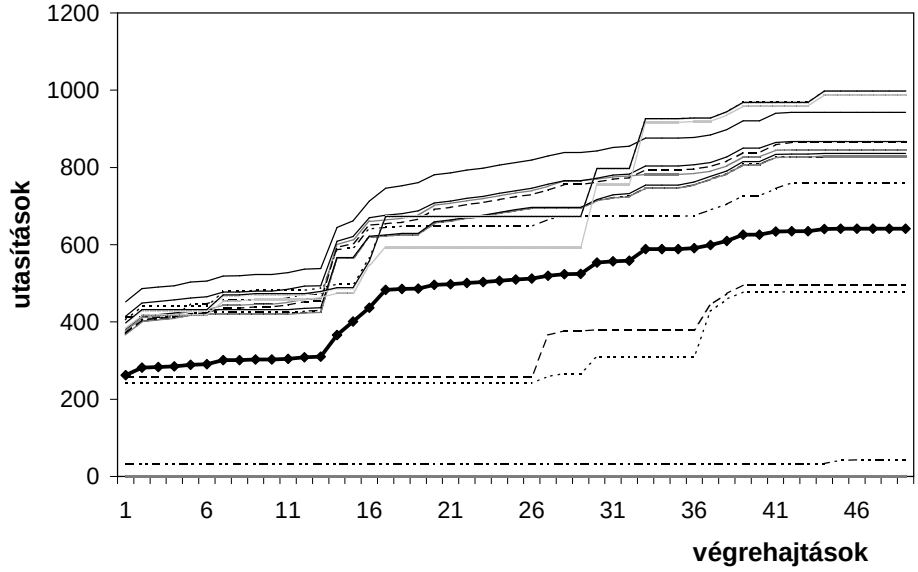
béket tüntettünk fel a jobb olvashatóság érdekében. A vastag vonalakkal a görbék átlagát jelöltük (minden kritériumhoz és nem csak a megjelenítettekhez). Látható, hogy a növekedés mértéke mindössze néhány (3–15) teszteset után csökkenni kezd. Természetesen, ebből merész lenne azt a következtetést levonni, hogy további tesztesetek hozzáadásával nem fog változni a tendencia, de valószínűsíthető, hogy a realizálható szeletet sokkal jobban nem közelítenénk már meg.

Másodszor azt vizsgáltuk meg, hogy a végső uniós szeletek méretei hogyan viszonyulnak a program, valamint ugyanazon kritériumokhoz kiszámított statikus szeletek méreteihez. Megjegyezzük, hogy ezen méréseket nem az utasítások száma, hanem a programsorok száma alapján végeztük, és az eredményeket is így mutatjuk be. Tudniillik, a statikus szeleteket kiszámító külső eszközzel csak így tudtunk összemérhető eredményeket gyártani. A 9.5. ábra ezen méréseink összesítő eredményét mutatja, ahol a programméretet, valamint az összes tesztesethez és kritériumhoz tartozó statikus és uniós szeletek méreteinek átlagát láthatjuk a három program esetében (végrehajtható utasításokat tartalmazó sorok számában, a 8. fejezetben az összes sorok számát adtuk meg). A programmérethez képest az összes statikus szelet aránya 72%, míg az uniós szeletek aránya 15% volt.

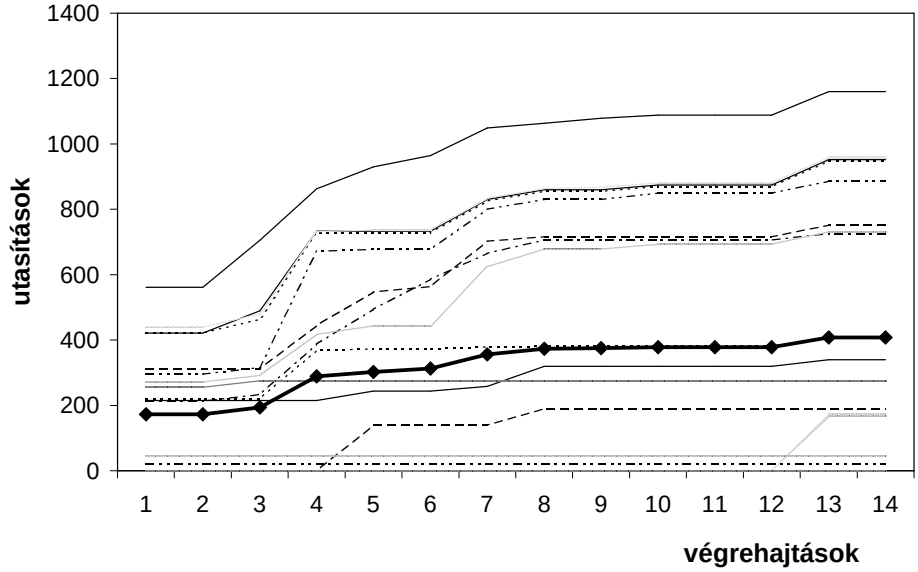
A szeletek méreteire vonatkozóan részletesebb adatokat szolgáltatnak a 9.6. ábra görbéi. Itt a statikus és uniós szeletek, a programméret százalékában kifejezett méreteinek eloszlását mutatjuk be. Az egyes méret-tartományok gyakoriságát hisztogramok formájában ábrázoltuk, a gyakoriságot az összes szelet-számhoz viszonyítva. A hisztogramokról azt



(a) *bzip*

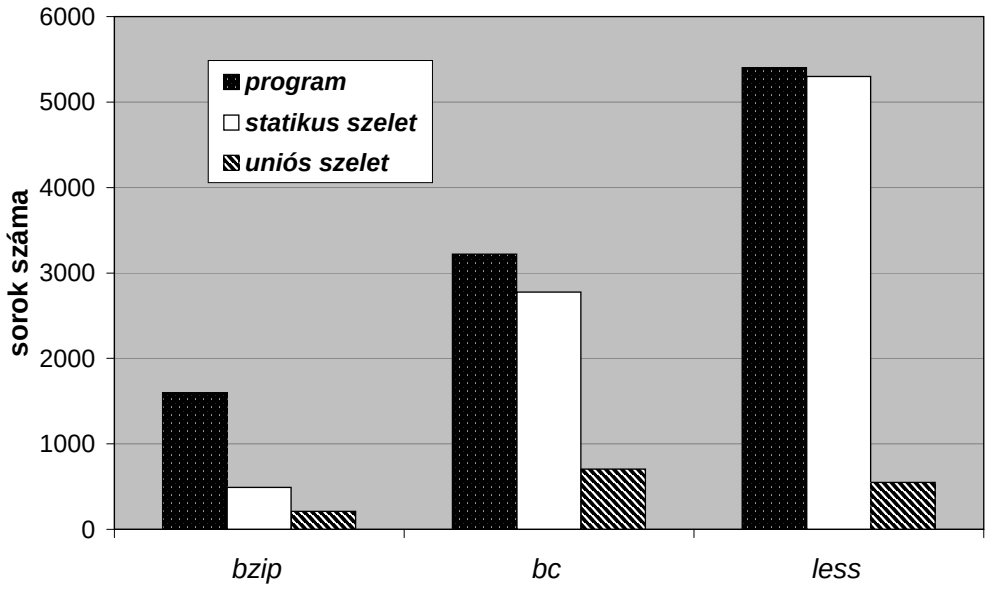


(b) *bc*



(c) *less*

9.4. ábra. Uniós szeletek növekedése



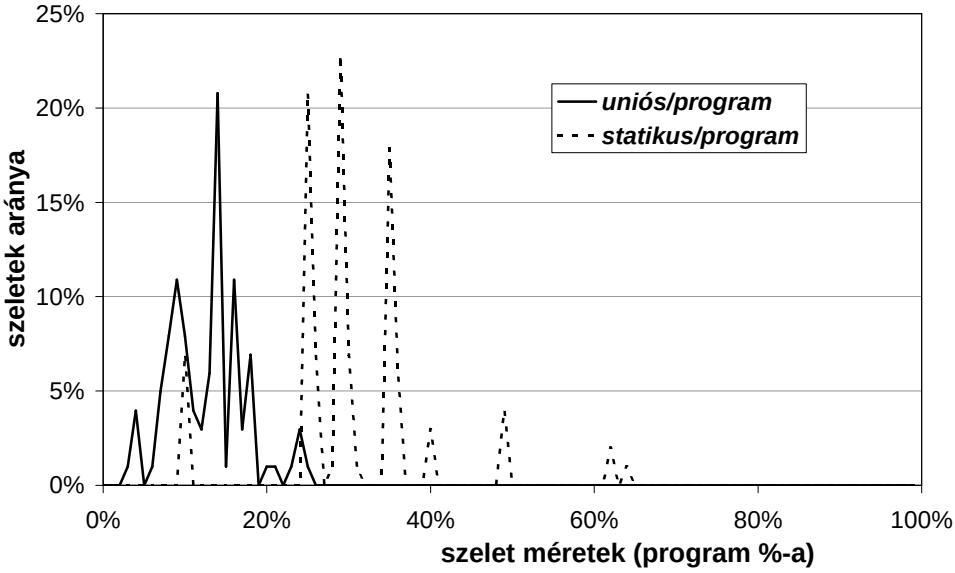
9.5. ábra. Átlagos szeletméretek

olvashatjuk le, hogy a három program meglehetősen különböző karakterisztikákat mutat, de az uniós szeletek általában lényegesen kisebbnek bizonyultak a megfelelő statikus szeleteknél. A *bzip* esetében körülbelül fele akkorák, míg a másik két program esetében a különbség sokkal lényegesebb. Ez abból adódik, hogy ezen programokra meglehetősen nagy statikus szeleteket kaptunk, sokszor majdnem a teljes programot. Ebből is látszik, hogy a statikus szeletek esetenként mennyire alkalmatlanok arra, hogy hasznos információt szolgáltassanak. Ezzel szemben az uniós szeleteink mindhárom program esetében a program mindössze 20–30%-át tették ki. Az egyenkénti dinamikus szeletméretek megfelelnek a a 8. fejezetben ismertetetteknek, nevezetesen a a programok megközelítőleg 5%-át képviselik.

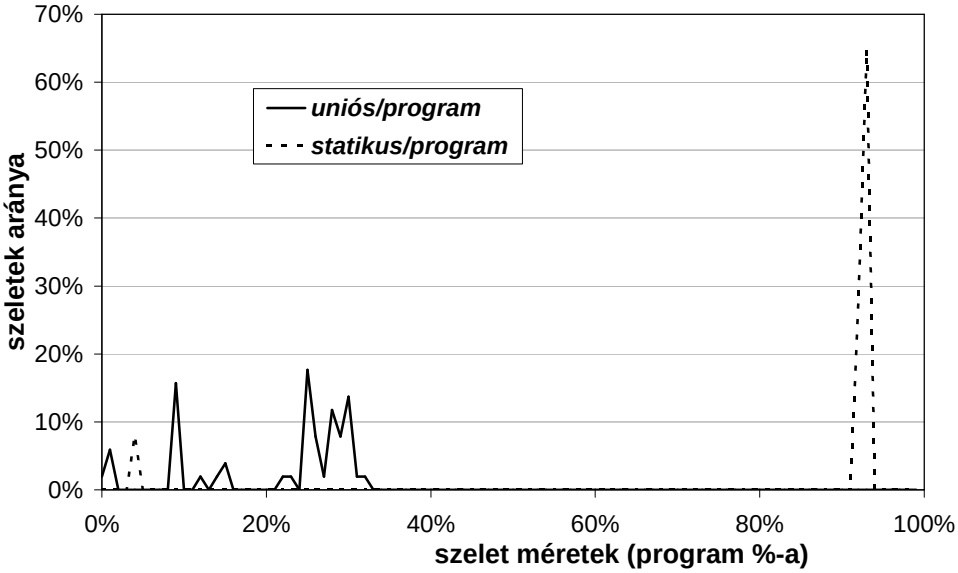
9.2.4. Uniós szeletek alkalmazása

Általában a szeletelési módszerek alkalmazása a szoftverkarbantartásban arra irányul, hogy a vizsgálandó programnak csak egy részét keljen megvizsgálni egy adott feladathoz (például egy adott programponton történt változás hatásai), és ne az egészet. A statikus szeletek használata egy módja a probléma-tér csökkentésének, de sokszor az így kapott méret-redukció túlságosan kicsi ahhoz, hogy valós segítséget nyújtson. Közismert, hogy a karbantartásra szánt erőforrások korlátozottak, így azok hatékony kihasználása nagyon fontos. Továbbá, a statikus szeletek, természetüknél fogva nem szolgáltatnak információt arra vonatkozóan, hogy a kapott szelet részeit milyen fontossági sorrendben érdemes kivizsgálni, melyek azon programrészek, amelyek valós használati eseteket tükröznek. Így az erőforrásokat sokszor a program olyan részeinek vizsgálatára fogjuk felhasználni, amelyek nem tükrözik a program valamely valódi használati esetéhez tartozó viszonyokat.

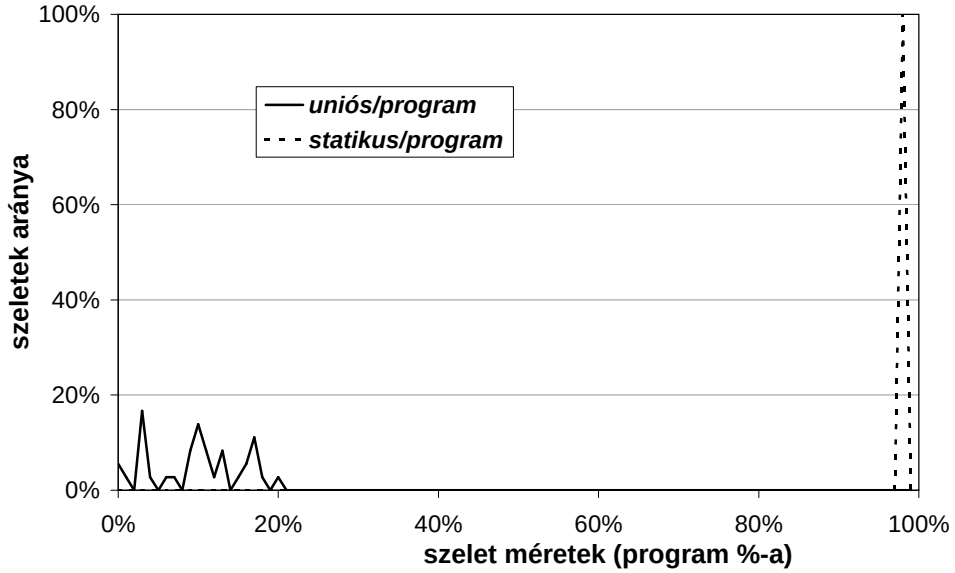
Az uniós szeletek használata pontosan ezt a problémát küszöböli ki azáltal, hogy rámutat a program azon részeire, amelyeket mindenféleképpen először kell megvizsgálni, hiszen azok valamely konkrét teszteset által kerültek bele a szeletbe. Ilymódon a karbantartást az uniós és statikus szeletek kombinált használatával lehet elvégezni: a folyamat kezdődhet a kiválasztott tesztesetekhez tartozó uniós szeletek vizsgálatával, amely lényegesen kevesebb erőforrást vesz igénybe, és sokszor elegendő az adott feladathoz. Majd, ha a feladat meg-



(a) *bzip*



(b) *bc*



(c) *less*

9.6. ábra. A szeletek méreteinek eloszlása

követeli a teljesen pontos vizsgálatot, a statikus és uniós szeletek különbségének vizsgálatát is el lehet később végezni.

A fent bemutatott méréseink azt igazolják, hogy a felvázolt megközelítés követhető, hiszen a kiindulási feltevéseink helyesnek bizonyultak. Nevezetesen, az uniós szeletek lényegesen kisebbek a statikus szeleteknél, valamint mindössze néhány reprezentatív tesztesettel elég jól közelíthetjük a realizálható szeletet, azaz a programnak azon részét, amelyet nagy valószínűséggel elegendő megvizsgálni.

Az uniós szeletek használata számos konkrét feladatnál jelent további segítséget, ahol elsődlegesen a statikus szeleteket javasolják. Ilyenek például a dekompozíciós szeletelés [41] és a szeletek által támogatott programmegértés [14] (ld. következő alfejezetet).

9.3. Egyéb lehetséges alkalmazások

Ebben az alfejezetben áttekintjük a dinamikus szeletek és az általunk kidolgozott algoritmusok további lehetséges alkalmazásait. Terveink között szerepel ezen alkalmazások részletesebb megvizsgálása és algoritmusaink kipróbálása ilyen környezetekben is. A teljesség igénye nélkül, ide tartoznak a program verifikáció [4, 49, 89], karbantartás [41], újratervezés [109], programmegértés [14, 50] és nyomkövetés [2, 57].

A verifikáció témakörében számos célra használják a szeletelési módszereket, például tesztelésnél automatikus teszt-adat generálásra, teszteset kiválasztásra, regressziós tesztelésre [4, 49, 57, 89]. Az általunk kifejlesztett globális dinamikus szeletelési algoritmus segítségével támogathatjuk a szeletelés alapú regressziós tesztelést, hiszen egy tesztesetre több szeletet határozhatunk meg egy feldolgozás során.

A szoftverkarbantartás egyik sarkalatos problémája annak biztosítása, hogy a szoftveren történt módosításnak minél kevesebb mellékhatása legyen. Erre a célra nagyon hasznos az ún. dekompozíciós szeletek alkalmazása [41]. Egy változóhoz tartozó dekompozíciós szelet a változó összes függőségét rögzíti, programponttól függetlenül. Egy változtatás után a dekompozíciós szelet megadja a változtatás által nem befolyásolt programrészeket, és ezáltal a regressziós tesztelést is megkönnyíti, hiszen az újratesztelendő programrészt csökkenti. Ilyen célra hagyományosan statikus szeletelést alkalmaznak, de uniós szeleteket is lehetne használni, hiszen ha egy adott változó összes előfordulásának uniós szeletéből meghatározzuk azok unióját, az így kapott uniós dekompozíciós szelet lényegesen kisebb lesz a statikus szeletnél. Ha eközben a globális algoritmusunkat használjuk, akkor egy futással több szeletet is meghatározhatunk.

A szoftverkarbantartás másik nagyon fontos tevékenysége a programkód átszervezése. Eközben a magas modul-kohézió elvárt tulajdonsága még egy sokszor módosított szoftvernek is. A statikus szeletelésen alapuló kohéziót mérő módszereknél lényegesen pontosabb eredményt szolgáltathatnak a dinamikus szeletek [44].

9.4. Összegzés

Láttuk, hogy a dinamikus szeletek nagy segítséget jelentenek számos programmegértési, verifikálási és szoftverkarbantartási probléma megoldásában. Ugyanakkor, a lehetséges alkalmazások köre még nyitott, hiszen ahhoz hogy ezen szoftver-analízis technika szélesebb körben elterjedjen, használható módszerekre van szükség. Az előző fejezetekben bemutatott algoritmusaink jó eséllyel indulhatnak ezen a pályán.

10. fejezet

Konklúziók

„Ez nem a vég, még csak nem is a vég kezdete.
De talán a kezdet vége.”

— Sir Winston Churchill

Közismert a szófordulat, mely szerint a kód dokumentálja saját magát. Ezt általában arra értik, hogy egy szoftver funkcionális és nem funkcionális tulajdonságai, szerkezeti és viselkedési architektúrája csak és kizárólag a programkód által van meghatározva, hiszen *az* maga a program. A magasabb szintű modellek, amelyek kísérhetik a kódot (kommentek, különböző tervezési modellek, dokumentáció) mind nagyon fontos kiegészítő információt hordoznak, de meglétükre és a kóddal való megfelelésükre nincsen garancia. Ezért minden olyan tevékenység, amely a létező rendszerek modellezésére, megértésére, újratervezésére, a verifikációra, hibajavításra, stb. – vonatkozik, a programkódból kell hogy kiinduljon. Ugyan a kód végső formája, amit a számítógép végre fog hajtani, a gépi kód, az egyértelmű megfeleltetése a magasabb szintű forráskódnak. Emiatt a forráskód analízise központi szerephez jut az említett tevékenységekben.

Az értekezés első részében ismertettünk egy olyan technológiát és megvalósult eszközt, amely nagy méretű, C/C++ nyelven íródott rendszerek forráskódjának analízisére alkalmas, a szoftver életciklusának bármely – elsősorban karbantartási – fázisában. Tudjuk, hogy a forráskód eredeti és végső feldolgozását a fordítóprogram fogja végezni, de annak egy konkrét célja van: előállítani a gépi kódot. Láttuk, hogy a forráskód analízise számos egyéb kontextusban is előtérbe kerül, ezért szükség volt olyan technológia kidolgozására, amely eleget tesz általánosabb követelményeknek is. A fordítóprogramokban használatos hagyományos megoldások sok tekintetben nem felelnek meg az említett céloknak, például nem állítanak elő részletes absztrakt modellt mely tükrözi a forráskódot, továbbá nem hibatűrők, részleges források elemzésére alkalmatlanok, stb. Az általunk kidolgozott analizálási technológia és a konkrétan megvalósult eszköz bizonyította alkalmasságát a fenti feladatok elvégzésére, számos alkalmazásban használják sikeresen szerte a világon. Ezen kívül, jelenleg is folyik a további lehetséges alkalmazások felkutatása és kidolgozása.

A dolgozat második részében a forráskód analízis egy konkrétabb területével foglalkoztunk, a programszeleteléssel. Különböző szeletelési módszereket ajánlanak a szoftverkarbantartás, programmegértés, verifikáció területein, hiszen e módszerek segítségével a feldolgozandó programot annak egy részére csökkenthetjük, és ezáltal a probléma méretét redukálhatjuk. Az eredeti megközelítés szerint egy program szeletét úgy kell érteni, hogy azon részét tartalmazza a programnak, amely kihatással lehet a kritériumra *valamely* futás során (statikus szelet). Az évek során a statikus szeleteléssel kapcsolatban jelentős

mennyiségű módszer, elméleti eredmény és konkrét eszköz látott napvilágot. Ugyanakkor a dinamikus szeletelés témaköre (amikor egy konkrét futás függőségeire vagyunk kíváncsiak) inkább háttérbe szorult, sokkal szerényebb a szakirodalma, a kevés számú publikált szeletelő eszköz hatékonysága nem meggyőző valós alkalmazásokban. Ennek egyik oka az lehet, hogy a létező algoritmusok költségesek, továbbá valós nyelvek szeletelésével kapcsolatban kevés információ lelhető fel. A dinamikus szeletek alkalmazása pedig számos területen előnyt jelent, hiszen azok méretei lényegesen kisebbek a statikus szeletekénél, és ezáltal pontosabb információt képesek szolgáltatni.

Kidolgoztunk két dinamikus szeletelési módszert, melyek hatékonysága lényegesen jobb mint a korábbi módszereké, alapvetően amiatt, hogy nem igényelnek nagy méretű függőségi gráfok tárolását és kezelését. A két módszer ugyanazon szeleteket számolja ki, de mások az előnyeik és hátrányaik. Működésük alapvetően különbözik egymástól, az első módszer globális, tehát az adott futáshoz tartozó összes dinamikus szeletet kiszámolja, míg a második igényvezérelt, tehát adott kritérium alapján dolgozik. Ezért különböző méretű programokra és felhasználásokra alkalmasak. Minden szeletelő módszer alapja egy megfelelő forráskód analizátor, a mi esetünkben az általunk kifejlesztett C/C++ analizátort használtuk fel, teljes sikerrel. A szeletelő algoritmusok teljes részletességgel lettek kidolgozva C nyelvű programok feldolgozására. A részleteket közzé is tettük, ami egyedülálló a dinamikus szeletelő módszerek körében. A prototípus implementációnkkal történt méréseinkkel, valamint konkrét alkalmazásokban is bizonyítottuk a szeletelő módszereink megfelelőségét, úgy mint nyomkövetés és nagy mennyiségű szeletek hatékony kiszámítása az ún. uniós szeletekhez. Jelenlegi implementációnk továbbfejlesztésén folyamatosan dolgozunk, hogy egy minden igényt kielégítő, optimalizált hatékonyságú, termék minőségű keretrendszert fejlesszünk ki.

10.1. Távlati tervek

Az ismertetett eredmények számos további kutatás kiindulópontját jelenthetik. A kifejlesztett C/C++ analizátor és a kapcsolódó keretrendszer alkalmazási köre korlátlan, amelyre néhány példát már láttunk a 4. fejezetben. Hasonlóan, a dinamikus szeletelő módszereket is több új alkalmazásban lehet majd felhasználni (ld. 9. fejezetet). A jelenlegi megvalósítás további fejlesztésekre, finomításokra is lehetőséget ad. Az alábbiakban ismertetünk néhány konkrét elképzelést, amivel a jövőben foglalkozni szeretnénk. Ezek közül számos már most kutatásunk tárgyát képezi.

10.1.1. C/C++ analizátor és keretrendszer

A Columbus keretrendszerrel számos további tervünk van, mint amilyen a szoftverek minőségi vizsgálatának elősegítése metrikák segítségével, valamint a SourceAudit-hoz hasonló cél-eszközök kifejlesztése a kigyűjtött forráskód modell felhasználásával. Érdekes kutatási területnek bizonyul a modell alapján a forráskód újragenerálása, amire modell-átalakítás után lehet szükség (az ún. *refactoring* végrehajtásakor a modell alapján felfedezett hiányosságokat javítjuk, és visszavezetjük a forráskódra [40]). Tervezzük továbbá, egyéb programozási nyelvek támogatását úgy front end, mint feldolgozó szinten (pl. Jáva). A front end kapcsán jelenleg is folyik a kutatómunka az előfeldolgozó és a C++ nyelvi sémák kompakt összekapcsolásán, amivel a felhasználást még könnyebbé tehetjük.

A CAN analizátor megvalósítását tovább szeretnénk javítani, hogy minél jobban támogassa a C++ szabványt, illetve további nyelvi dialektusokat. A javítás magában foglalja például a C++ sablonok teljesebb kezelését. Ami az elemző további felhasználását illeti, már említettük egy kísérleti fordítóprogram és egy forráskód nyelv-közti fordító (tolmácsoló) kifejlesztését, valamint a dinamikus hívási gráfokkal kapcsolatos vizsgálataink folytatását.

10.1.2. Dinamikus szeletelés

Az első továbbfejlesztés a szeletelő módszereinkkel kapcsolatban az, hogy azokat valódi keretrendszerbe építsük be, amelyben az algoritmusok implementációi optimalizáltak, a valós használati környezetekhez igazítottak. Az alkalmazás történhet a Columbus rendszeren belül, de különálló eszközként is. A nyomkövetés alkalmazásban nyilvánvalóan a futtató környezetbe kell beépítenünk a módszereket. Már elkezdtük a megvalósítást a GCC fordítóprogramhoz kötődő GDB nyomkövetőn belül, amely egy online megvalósítása a módszereknek. Mindkét algoritmust megvalósítjuk annak érdekében, hogy kísérleti úton megállapítsuk azok alkalmazhatóságát e környezetben. Az eredményeket természetesen nyílt forráskódként elérhetővé tesszük, és megkíséreljük a beépíttetését a hivatalos verziókba is.

Ezen célok eléréséhez további feladatok várnak ránk: az implementáció tesztelése minél több szoftveren, speciális dialektusok és nyelvi szolgáltatások kezelése, és az algoritmusok optimalizálása. Utóbbira néhány lehetőséget már tárgyaltunk a 6. és 7. fejezetekben. Lehetőség van továbbá a végrehajtási nyom hatékonyabb feldolgozására, például úgy, hogy azt tömörítve tároljuk (ilyen megoldást javasolnak egyes kutatók, ld. például [102]). Ezen kívül Zhang *és mások* által javasolt gyorsítási lehetőségek alkalmazhatóságát is megvizsgáljuk az algoritmusainkra [105, 106, 107]. Az egyik módszer szerint a végrehajtási nyomon korlátozott előfeldolgozást kell alkalmazni, mely során összesített információkat tárolunk el adott időközönként és azokat újrafelhasználjuk egy későbbi szeletelési igény esetén.

Végül, tervezzük a módszerek kiterjesztését objektumorientált nyelvek szeletelésére is. Jelenleg folyik a vizsgálat a C++ szeletelésével kapcsolatos problémák és megoldási lehetőségek feltérképezésén.

A. Függelék

Magyar nyelvű összefoglaló

„A tudás kezdete egy olyan dolognak a felfedezése, amit nem értünk.”
— Frank Herbert

1. Bevezetés

Az értekezés kiindulási pontja az, hogy a szoftverrendszerek forráskódjának ember által történő megértése, értelmezése merőben másfajta problémákat vet fel, mint ugyanannak gépi feldolgozása, pontosabban mondva a fordítóprogram által történő analizálás. A megértés nem könnyű feladat, főleg ha figyelembe vesszük a szoftverek komplexitását és méretét, valamint azt, hogy sokszor ismeretlen kóddal kell szembenézni. A kód értelmezésének igénye az életciklus szinte bármely pontján felmerülhet, úgy mint nyomkövetéskor és verifikáció utáni hibajavításkor, vagy az evolúciós fázisban, amikor a karbantartáshoz szükséges információk sokszor egyetlen forrása maga a forráskód. A forráskód megértését különböző módon támogathatjuk szoftver-eszközökkel, mivel számos analizálási feladatot automatikusan is elvégeztethetünk. Ehhez általában valamilyen magasabb absztrakciós szintű modelleket állítunk elő a *forráskód analizátorok* segítségével, amely modellek alkalmasabbak emberi feldolgozásra (többek között ezt nevezzük *reverse engineering*-nek [20]). A fordítóprogram azon részének, amely elkezdi a forráskód feldolgozását (a *front end*-nek) hasonló feladata van, nevezetesen az a szintaktikus elemzés után előállítja a forrás valamilyen belső reprezentációját. Ugyanakkor a kettőnek merőben más céljaik vannak, ezért a fordítóprogram általában nem használható fel más analizálási feladatokra. Ez eredményezte a speciális analizálási technológiák kidolgozását és eszközök kifejlesztését.

Mi is elkezdtük egy technológia- és eszközrendszer kidolgozását a programmegértés támogatásának céljával. A jelen értekezés szerzője e környezet két fontos elemét dolgozta ki. Az egyik a minden analizálási feladat alapja, egy általános célú C/C++ forráskód analizátor front end, amely meghatározott sémának megfelelő modellt állít elő. Továbbá, a front end a fordítóprogramok front end-jeihez képest számos speciális tulajdonsággal bír, amelyek nélkülözhetetlenek a megértéshez. A másik eredmény a dinamikus programszeletelő, egy speciális kód-analizálási eszköz, amely két algoritmust valósít meg, melyek a korábbi módszerekhez képest lényegesen hatékonyabbak. Az algoritmusokat a korábbiakhoz képest sokkal részletesebb leírással láttuk el ami a valós nyelvek analízisét illeti. Ezen összefoglaló e két téma legfontosabb eredményeit tekinti át egy rövid konklúzióval a végén.

2. C/C++ forráskód analízis és modell-építés

(*A szerző kapcsolódó eredményei megtalálhatók a [29], [34] és [101] publikációkban.*)

Saját front end kidolgozását két tényező is befolyásolta. Először is, azon analizátor front end-ek, melyek a C++ nyelvet megfelelő módon analizálják, gyakorlatilag megfizethetetlenek, azokból is mindössze egy-két van jelen a piacon. Ez nyilvánvalóan annak a következménye, hogy a nyelv rendkívül nehezen analizálható. A másik csoport, tehát az elérhető megoldások viszont nem alkalmasak céljainkra, hiszen az általános célú front end-eknek támasztott követelményeket nem teljesítik maradéktalanul. Egy általános front end sok mindenben eltér a fordítóprogramokban lévőktől, amely különbségek közül a legfontosabbak itt megemlítjük (ld. a [34] cikket). Először is az elemző teljes elemzést kell hogy végezzen, hiszen minden elérhető információt tárolni kell. Ezzel szemben számos egyéb megoldás részleges tényfeltárást végez, hiszen korlátozott számú analízist támogatnak. Továbbá, az elemzőnek hibatűrőnek kell lennie azért, hogy a lehető legtöbb információt gyűjtse ki nem teljes vagy hibás kódból is. Ugyancsak fontos a nyelvi dialektusok kezelése (esetleg a felismert nyelv bővíthetősége), és a más eszközökkel való integráció könnyűsége (például jó csere-formátum és parancssori működés).

Számos megoldással találkozhatunk az elérhető C/C++ front end-eknél. Néhány közülük fordítóprogramokon alapul, ezért azok nem alkalmasak általános célú felhasználásra. Továbbá, azon analizátorok, melyek integrált fejlesztőkörnyezetek vagy általános programmegértést támogató eszközök részei sokszor részleges információk kigyűjtését végzik vagy nem rendelkeznek megfelelő interfészekkel az adatcserét elősegítendő. Ezért fejlesztettünk ki saját általános front end-et a C/C++ nyelvekhez.

A mi megoldásunk két fő eszközből áll, egy előfeldolgozóból és egy nyelvi analizátorból. Mindkettő előállítja a forráskód hozzátartozó modelljét a megfelelő sémák szerint (ld. [29] és [101] cikkeket). A nyelvi elemzőben alkalmazott elemzési technológia felülről lefelé haladó, erősen $LL(k)$-alapú, rekurzív alászálló eljárásokkal, amit a nyelv megalkotói is javasolnak [95] és néhány fordítóprogram is alkalmaz. Az elemzőt elemzőgenerátorral készítjük, amely hatékony megoldásokat is nyújt a C++ nyelvi kétértelműségek kezelésére. Az elemzéssel párhuzamosan építjük a belső reprezentációt szintaxis-vezérelt módon. A belső modell egy absztrakt szintaxis gráf (ASG), amit az elemzési fából származtatunk, kiegészítve további kapcsolatokkal a nyelvi elemek között. Ugyancsak az ASG-t használjuk szimbólum-tábla gyanánt az elemzés kétértelműségi szituációinak feloldására.

Az analizátor front end-nek sajátos tulajdonságai vannak, amelyekkel a fent említett követelményeknek teszünk eleget. Például a hibatűrést kifinomult, heurisztikákon alapuló hiba-helyreállító módszerrel oldjuk meg. Továbbá, az analizálási sebességen az ún. előfeldolgozott fejlécfájlokkal és az elemzésből és modellépítésből kizárt nyelvi elemekkel (például függvény-törzsek) gyorsíthatunk.

A front end használhatóságát több valós méretű rendszer sikeres analízise igazolja. Mind az analizálás sebessége, mind a felhasznált memória alkalmasak nagy méretű rendszerekhez is. Például a [34] publikációban bemutattuk, hogy a memória fogyasztás közel lineáris a program méretével, továbbá, az analizálási sebesség a fordítóprograméval összemérhető. A front end legfontosabb alkalmazása a Columbus keretrendszeren belül van [33, 34], amely teljes megoldást nyújt a rendszerek elemzéséhez, magában foglalva a projekt kezelést, modellek összefésülését, szűrését és transzformálását különböző formátumokba. Utóbbi ma-

gában foglal különböző XML alapú, a sémánknak és egyéb csere-formátumoknak megfelelő modelleket, valamint további származtatott eredményeket, mint például UML modellek és metrikák.

A másik fontos alkalmazása a front end-nek a dinamikus szeletelő eszköz statikus analizátoraként van, amit a következő alfejezetben tekintünk át.

3. Dinamikus programszeletelés

(*A szerző kapcsolódó eredményei megtalálhatók a [8], [12] és [46] publikációkban.*)

A programszeletelés [97, 103] olyan program-analizálási technika, amelyet számos szoftverfejlesztési területen javasolnak, mint amilyen a verifikáció és karbantartás. Jelen értekezés témája a *hátrafelé irányuló dinamikus szeletelés*, amely az analizálandó program egy részhalmazát határozza meg azon utasításokkal, melyek kihatással voltak egy adott program-pont adott változó-előfordulásainak értékeire a program egy konkrét futása esetén (szemben a statikus szeleteléssel, amikor minden lehetséges végrehajtást figyelembe veszünk). A korábbi, dinamikus szeletek meghatározását végző módszerek nem alkalmasak valódi méretű programok analízisére komplexitásuk miatt. A legfontosabb módszer, a DDG gráfokon alapuló [3] olyan gráfokat használ akciók (utasítás előfordulások a végrehajtásban) közötti függőségekkel, melyek mérete korlátlan (a végrehajtott lépések számával van meghatározva).

Két dinamikus szeletelő módszert dolgoztunk ki, amelyek a statikus fázisban előállított lokális definiálás-használat információk alapján működnek. E reprezentáció rögzíti minden utasításban a definiált és a használt változók neveit, a vezérlési függőségekkel egyetemben, amelyeket a virtuális, ún. *predikátum változókkal* határozunk meg. A programreprezentáció tárolása és használata lényegesen kisebb költségeket jelent, mint az egyéb módszerek esetében használatos megoldások. A két algoritmus abban tér el egymástól, ahogy a végrehajtási nyomot (futtatási lépések története) dolgozzák fel. Az első *globális*, amely a nyomot az első végrehajtott utasítással kezdve olvassa be, és minden lépésnél meghatározza az aktuális utasításban definiált változóhoz tartozó függőségi halmazt, amihez a használt változók legutóbb kiszámolt függőségi halmazait és a legutóbbi definiáló utasításait használja fel (ld. [12, 46] cikkeket). Ilymódon minden dinamikus szeletet megkapunk és kimenetre adunk a definiált változókhoz az utasítások minden előfordulásánál (a memóriában csak az érvényben lévő halmazokat tároljuk). Ez a módszer ott alkalmas, ahol több szelet kiszámítására is szükség van a végrehajtási nyom egyszeri feldolgozásával. A második algoritmus *igényvezérelt*, ami azt jelenti, hogy egyetlen szeletet határoz meg egy kérés alapján, és minden egyes igény esetén a nyomot újra feldolgozza a kiindulási utasítással kezdve, hátrafelé haladva és úgy követve a függőségeket. Az algoritmus bejárja a dinamikus függőségeket és a még fel nem dolgozott akciókat egy munkahalmazba gyűjti. Amikor minden függőség fel lett oldva és a munkahalmaz kiürül, akkor fejezi be a számítást. Mindkét algoritmusnak megvannak az előnyeik és hátrányaik, ezért az alkalmazási köreik is különbözőek.

Minden szükséges részletet megadunk az algoritmusok megvalósításához valós procedurális nyelvekre, valamint konkrét megvalósítást nyújtunk C programok szeletelésére. Ez magában foglalja a függőségek memória cellákon történő követését általános módon, ahelyett hogy a különböző entitásokat különbözőképpen kezelnénk. Ez azt jelenti, hogy minden változót és memória hivatkozást (mint például mutató indirekció) konkrét memória címekké

alakítunk át a nyom feldolgozásakor, majd ezeken végezzük a függőségek számítását. Megoldást nyújtunk a tetszőleges vezérlés-átadások, interprocedurális függőségek, összetett adattípusok, stb. kezelésére. Ezen részleteket először a [12] cikkben publikáltuk, amely elnyerte a bemutatást fogadó konferencia („European Conference on Software Maintenance") legjobb munkájáért járó díjat.

Méréseink alapján az algoritmusok futási ideje és memória fogyasztása lényegesen jobb a legrosszabb esetekre meghatározott komplexitásokhoz képest. Nevezetesen, a legfontosabb tényezők vagy a program mérettel, vagy pedig a használt különböző memória címek számával vannak összefüggésben, és nem pedig a végrehajtott lépések számával.

A dinamikus szeletek legfontosabb alkalmazása a nyomkövetésnél van. Ehhez megadtunk egy kiegészítést, amely az ún. *releváns szeleteket* határozza meg, amelyek jobb eredményt szolgáltatnak egyes kód-szerkezetekre [46]. Azonban a dinamikus szeletelés szélesebb körben is alkalmazható, például a globális algoritmusunkat használva sok szelet határozható meg hatékonyan, és emiatt számos karbantartási és tesztelési feladat támogatható segítségével. Egy konkrét módszert is megadtunk, amellyel az ún. *uniós szeleteket* határozhatjuk meg, ahol a a program különböző végrehajtásaihoz tartozó dinamikus szeletek egyesítését számoljuk ki [8]. Az uniós szelet nagyobb mint bármely dinamikus szelet, viszont lényegesen kisebb a statikus szeletnél, tehát pontosabb is annál. Ennek az a haszna, hogy az uniós szeleteket számos statikus szeleten alapuló alkalmazásban használhatjuk fel utóbbiak helyett. Kísérleteink eredményei azt mutatják, hogy a dinamikus szeletek általában kicsik (a program megközelítőleg 5%-a), míg a statikus szeletek lényegesen nagyobbak (általában több mint a program 70%-át teszik ki). Ugyanakkor a már kevés számú, reprezentatív végrehajtással kiszámított uniós szelet növekedési sebessége is lényegesen lecsökken, miközben a mérete jelentősen kisebb lesz a statikus szeletnél (a program hozzávetőlegesen 15%-a).

4. Konklúziók

A szoftverrendszerek egyetlen megbízható reprezentációja sok esetben a forráskód maga, ezért annak ember által történő megértésének támogatása kulcsfontosságú. E munkában a megértést két módon támogatjuk. Először, egy általános C/C++ analizátor front endet nyújtunk, amely a neki támasztott speciális követelményeknek megfelel. Másodszor, két hatékony dinamikus szeletelési módszert adunk meg minden részlettel C programok analíziséhez, melyek alkalmasak valós méretű rendszerek feldolgozására. A jövőben tervezzük a módszerek valódi környezetbe történő beépítését, úgy mint a nyomkövető, valamint szeretnénk megadni az objektumorientált, elsősorban C++ nyelvű programok dinamikus szeletelésére vonatkozó megoldásokat.

B. Függelék

Summary in English

"The beginning of knowledge is the discovery of something we do not understand."

— Frank Herbert

1 Introduction

The starting point of the thesis is that the human comprehension of a software system's source code is quite different from the machine's understanding (the compiler, to be more precise). The former may be a very difficult task because of the size and complexity of the software, and the unfamiliarity with it of the person involved. The need for program comprehension can arise in many situations throughout the software life cycle, such as during debugging and error correction after verification, or in the phase of evolution when, for software maintenance, the only reliable source of information is the source code itself. Code comprehension can be aided by software tools in various ways because many analysis tasks can be performed automatically. The general approach is to produce some higher level models from the code that are more suitable for human processing using the so-called *source analyzers* (this is part of *reverse engineering* [20]). The first part of a compiler dealing with the source code (the *front end*) performs similar tasks, namely it parses the code and produces some internal representation of it. However, the two have different purposes, and so the compiler cannot be generally used for other code analysis tasks. Hence the need for special code analysis technologies and analyzers.

We have started to develop and do research within a framework that consists of basic technologies and software tools to aid code comprehension. The author of this thesis elaborated two important aspects of the framework. The first one is the basis of all analysis tasks, a general C/C++ analyzer front end. It produces models of the software according to a well defined schema, and it provides the kind of features that the compiler front end does not need to address as special requirements for the task of comprehension. The other result is a special code analysis tool, a dynamic program slicer. The tool implements two efficient dynamic slicing algorithms that represent novel techniques, and compared to previous solutions are significantly more efficient. Furthermore, implementation peculiarities are provided for a real programming language in much more detail than anywhere previously available. The remainder of this summary outlines the main results of these two topics in separate sections with a short conclusion section.

2 C/C++ Source Code Analysis and Model Creation

(*Related results of the author can be found in [29], [34] and [101].*)

There were two motivations for developing a front end. The first one is pragmatic: the analysis of the C++ language is very hard, and a good, general purpose analyzer front end is quite expensive (in fact, there are only a few available on the market). The reachable solutions on the other hand, are not really suitable for our purposes as they do not satisfy the criteria for a general purpose front end. A general front end differs in many ways from that for a compiler. Here is a list of the most important aspects (see [34] for more). First, the parser needs to perform a complete analysis of the source, since all the available information needs to be preserved. This is in contrast with a number of available solutions where only a partial extraction of facts is done because they only need to deal with a limited number of analyses. Next, the parser needs to be fault tolerant to extract as much information as possible from incomplete or erroneous code. In addition, important aspects are the handling of language dialects (possibly with extendibility of the accepted language), and the ease of integration into analyzer tools (with good exchange formats and preferably a command-line operation).

For the C/C++ languages there are a number of related solutions to analyzer front ends. Some of them are based on compiler technology, therefore these are inappropriate. Furthermore, the analyzers that are part of integrated development environments or more general code comprehension tools and frameworks usually extract only partial information and do not provide suitable interfaces for data exchange. Hence we developed a general front end for C/C++.

It is composed of two major parts: a preprocessor and a language analyzer. Both produce the appropriate models of the source code according to the prescribed schemas (see [29] and [101]). In the language analyzer the parsing technology employed is top-down parsing with recursive descent procedures based on strong $LL(k)$, which was also originally suggested by the language creators [95] and is used by several compilers as well. The parser is generated using a parser generator, which provides efficient techniques for handling language ambiguities (which C++ is full of). During the parsing the internal representation is built in a syntax-directed manner. The internal model that is built-up is an Abstract Syntax Graph or ASG, which means that the relevant syntactic constructs are derived from the parse tree with additional relations among the language elements. Since the ASG also serves as a symbol table, it is used for disambiguation of parsing at the relevant places.

The analyzer front end has just the features for meeting the requirements mentioned above. For example, the fault tolerance is addressed by employing a sophisticated *error recovery* technique based on heuristics, and the analysis speed can be improved by using the so-called *precompiled headers* and excluding certain elements from complete parsing and model building (like method bodies).

The usefulness of the front end has been proved in several cases with real size systems. The analysis speed and the memory consumed are both suitable for very large systems as well. For example, in [34] we showed that the memory consumption is nearly linear with the size of the program, while the analysis time is comparable to the compiler's. The most important application of the C++ analyzer front end is within the Columbus framework [33, 34], which provides a complete analysis of systems, including project handling, model

merging, filtering and transformation to various formats. The formats include XML-based models according to the schema and other common exchange formats, as well as further transformations like UML models and metrics.

The other important application of the analyzer is its use as the static front end of the dynamic slicing tool, which is outlined in the next section.

3 Dynamic Program Slicing

(*Related results of the author can be found in [8], [12] and [46].*)

Program slicing [97, 103] is a program analysis technique proposed for many software engineering fields including verification and maintenance. In this thesis we deal with *backward dynamic slicing*, whose purpose is to determine a subset of the subject program consisting of statements that actually influenced the values of the variables at a specific program point and in a specific execution of the program (in contrast, with static slicing we are interested in all possible executions). The previously available methods for determining dynamic slices are inappropriate for real size applications because of their complexity. The most important method, based on DDG graphs [3] requires graphs with dynamic dependences among the actions (instruction occurrences in the execution) of unbounded size (determined by the number of instructions executed).

We introduced two methods for computing dynamic slices based on local definition-use information built in a static analysis phase. It records the name of variables defined and used at each instruction, incorporating also the control dependences by using the artificial, so-called *predicate variables*. The storage and use of this program representation introduces much less overheads compared to the previous approaches. The two algorithms differ in the way the execution trace is processed. The first one is the *global* where it reads the trace from the first instruction executed and at each step it computes the dependence set for the defined variable at the actual instruction based on the previously computed dependence sets of the used variables and the instruction where they were last defined (see [12, 46]). This way all the dynamic slices corresponding to defined variables at all occurrences of instructions are gained and written to the output (only the actually effective sets are stored in the memory). This algorithm is suitable for applications where all slices need to be computed with one pass through the execution trace. The second algorithm is *demand driven*, meaning that it computes only one slice per request and it processes the trace each time starting with the statement for which the dependences should be computed in a backward fashion. The algorithm scans through all dynamic dependences and tracks the unprocessed actions in a worklist. It terminates when all dependences have been processed transitively and the worklist becomes empty. Both algorithms have their advantages and drawbacks, hence their fields of application are also different.

We provided all the details for implementing the algorithms for procedural languages; concrete solutions are given for C. This includes the handling of dependences among memory locations in a general way rather than treating different kinds of entities differently. This implies that all variables and memory dereferences (like pointer indirections) are transformed to actual memory addresses during processing the trace, and then dynamic dependences are computed based on these. We give solutions for handling arbitrary control flows, interpro-

cedural dependences, composite data types and so on. These details were first published in [12], which was considered to be the best paper of the conference (the European Conference on Software Maintenance).

According to the measurements carried out, the execution time and memory consumption of the algorithms is much better than the complexities determined for the worst cases. Namely, all significant factors are correlated with the program size or the number of different memory locations used by the program, rather than the number of instructions executed.

The most important application of dynamic slices is in debugging. For this we provided an extension for computing the so-called *relevant slices*, which provide better results for certain code constructs [46]. However, the application range of dynamic slicing is much wider, for instance using our global algorithm many slices can be computed efficiently, so many maintenance and testing tasks can be aided. As a concrete application we elaborated on the method for computing the *union slices*, where the union of dynamic slices is determined for different executions of the program [8]. The union slice is bigger than any individual dynamic slice but is significantly smaller than the static slice, hence it is more precise. The effect is that they can be used in many applications instead of the static slice. The results of our experiments showed that the dynamic slices are generally small (about 5% of the program), while the static slices are much larger (more than 70% of the program on average). However, the union slices computed for a small number of representative executions are small enough (about 15% of the program), and the size does not increase significantly by adding new executions.

4 Conclusions

In many cases the source code is the ultimate representation of a software system, therefore to aid its comprehension by humans is very important. This work addressed the problem of code comprehension in two ways. First, by providing a general C/C++ analyzer front end which meets the special requirements for them. Second, two efficient dynamic slicing methods were elaborated on for analyzing C programs, which are suitable for analyzing real life systems. In the future we plan to integrate the methods into real environments (like a debugger) and also provide the solutions for dynamic slicing object-oriented, C++ programs.

Irodalomjegyzék

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

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software – Practice and Experience (SPE)*, 23(6):589–616, 1993.

[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] The ANTLR C++ grammar web site. `http://www.antlr.org/grammar/cpp`.

[6] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from C++ source code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, September 2003.

[7] Bell Canada Inc., Montréal, Canada. *DATRIX – Abstract semantic graph reference manual*, version 1.4 edition, May 2000.

[8] 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.

[9] Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. CSiBE benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7–15, June 2004.

[10] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, September 2003.

[11] Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. Optimizing for space : Measurements and possibilities for improvement. In *Proceedings of the 2003 GCC Developers' Summit*, pages 7–20, May 2003.

[12] 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 Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 2001.

[13] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04)*, pages 43–52. IEEE Computer Society, September 2004.

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

[15] The Borland Together web site. `http://www.borland.com/together/`.

[16] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[17] The Cetus web site. `http://paramount.www.ecn.purdue.edu/ParaMount/Cetus/`.

[18] Jiun-Liang Chen, Feng-Jian Wang, and Yung-Lin Chen. Slicing object-oriented programs. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC'97/ICSC'97)*, pages 395–404, Hong Kong, December 1997.

[19] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *ACM SIGPLAN Notices*, pages 144–153, October 1995.

[20] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[21] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the 1997 International Conference on Software Maintenance (ICSM)*, pages 188–195, Bari, Italy, October 1997.

[22] Sebastian Danicic, Andrea De Lucia, and Mark Harman. Building executable union slices using conditioned slicing. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 89–97, Bari, Italy, June 2004.

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

[24] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of WCRE'01*, pages 59–67, October 2001.

[25] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, pages 175–186, October 1999.

[26] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of Languages and Compilers for Parallel Computing, 5th International Workshop*, volume 757 of *Lecture Notes in Computer Science*, pages 497–511, New Haven, Connecticut, USA, August 1992.

[27] J. Ebert, R. Gimnich, H. H. Stasch, and A. Winter. GUPRO – generische umgebung zum programmverstehen, 1998.

[28] The EDG web site. `http://www.edg.com/`.

[29] Rudolf Ferenc and Árpád Beszédes. Data exchange with the Columbus schema for C++. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, March 2002.

[30] Rudolf Ferenc and Árpád Beszédes. Az objektumvezérelt szoftverek elemzése. In *VIII. Országos (Centenáriumi) Neumann Kongresszus Előadások és Összefoglalók*, pages 463–474. Neumann János Számítógép-tudományi Társaság, October 2003.

[31] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Extracting facts with Columbus from C++ code. In *Tool Demonstrations of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8, March 2004.

[32] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact extraction and code auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 513–513. IEEE Computer Society, September 2004.

[33] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[34] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[35] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing design patterns in C++ programs with the integration of Columbus and Maisa. In *Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 58–70. University of Szeged, June 2001.

[36] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus – tool for reverse engineering large object oriented software systems. In *Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[37] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from open source software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[38] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a standard schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, October 2001.

[39] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[40] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Pub Co, June 1999.

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

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[43] R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the Conference on Software Maintenance*, pages 191–200, Sorrento, Italy, 1991. IEEE Computer Society Press.

[44] Neelam Gupta and Praveen Rao. Program execution-based module cohesion measurement. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 144–153, November 2001.

[45] Rajiv Gupta and Mary Lou Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 29–40, October 1995.

[46] 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.

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

[48] Mark Harman, David W. Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software (JSS)*, 68(1):45–64, October 2003.

[49] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification & Reliability (STVR)*, 5(3):143–162, September 1995.

[50] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Proceedings of 5th International Workshop on Program Comprehension*, pages 70–79, Dearborn, Michigan, USA, 1997.

[51] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Florence, Italy, November 2001.

[52] Tommy Hoffner, Mariam Kamkar, and Peter Fritzson. Evaluation of program slicing tools. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 51–69, Saint Malo, France, May 1995.

[53] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171, November 2000.

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

[55] International Standards Organization. *Information technology – Syntactic metalanguage – Extended BNF*, ISO/IEC 14977:1996 edition, 1996.

[56] International Standards Organization. *Programming languages – C++*, ISO/IEC 14882:2003 edition, 2003.

[57] Mariam Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linköping University, 1993.

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

[59] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming*, 38:625–636, 1993.

[60] 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.

[61] 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.

[62] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[63] Bogdan Korel. Computation of dynamic slices for programs with arbitrary controlflow. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 71–86, Saint Malo, France, May 1995.

[64] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, January 1997.

[65] Bogdan Korel and Roger Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, 2(2):199–215, 1992.

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

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

[68] Bogdan Korel and Juergen Rilling. Application of dynamic slicing in program debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG)*, pages 43–58, Linköping, Sweden, May 1997.

[69] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, Washington, August 1994.

[70] Jens Krinke. Static slicing of threaded programs. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'98)*, number 33(7) in SIGPLAN Notices, pages 35–42, Montreal, Canada, June 1998.

[71] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, 1996.

[72] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. *Languages and Compilers for Parallel Computing, 16th International Workshop (LCPC 2003)*, 2958:539–553, 2004.

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

[74] Brian A. Malloy, Tanton H. Gibbs, and James F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software – Practice and Experience (SPE)*, 33(1):19–39, January 2003.

[75] E. Mamas and Kostas Kontogiannis. Towards portable source code representations using XML. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 172–182, November 2000.

[76] Jean Mayrand and François Coallier. System acquisition based on product assessment. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, pages 210–219, March 1996.

[77] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, February 1998.

[78] Hausi A Müller, Kenny Wong, and Scott R Tilley. Understanding software systems using reverse engineering technology. In *Proceedings of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*, 1994.

[79] Object Management Group Inc. *OMG Unified Modeling Language Specification*, version 1.5, 2003.

[80] The OpenC++ web site. `http://www.csg.is.titech.ac.jp/~chiba/openc++.html`.

[81] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, number 19(5) in SIGPLAN Notices, pages 177–184, Pittsburgh, Pennsylvania, May 1984.

[82] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995.

[83] The PCCTS web site. `http://www.antlr.org/pccts133.html`.

[84] James F. Power and Brian A. Malloy. An approach for modeling the name lookup problem in the C++ programming language. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2000)*, pages 792–796, March 2000.

[85] The Rational Rose web site. `http://www-306.ibm.com/software/rational`.

[86] The Rigi web site. `http://www.rigi.csc.uvic.ca`.

[87] Juergen Rilling and Bhaskar Karanth. A hybrid program slicing framework. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 12–23, November 2001.

[88] Claudio Riva, Michael Przybilski, and Kai Koskimies. Environment for software assessment. In *Proceedings of ECOOP'99 Workshop on Object-Oriented Architectural Evolution*, 1999.

[89] 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.

[90] Sander, G. VCG Overview. `http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html`.

[91] The Semantic Designs web site. `http://www.semdesigns.com/`.

[92] The SNiFF+ web site. `http://www.windriver.com/products/development_tools/ide/sniff_plus/`.

[93] The Source-Navigator IDE web site. `http://sources.redhat.com/sourcenav`.

[94] The Spirit web site. `http://spirit.sourceforge.net/`.

[95] Bjarne Stroustrup. *The Design and Evolution of C++*. AT&T Bell Labs, 1994.

[96] Gyöngyi Szilágyi, Tibor Gyimóthy, and Jan Maluszynski. Static and dynamic slicing of constraint logic programs. *Automated Software Engineering*, 9(1):41–65, January 2002.

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

[98] Masaru Tomita. *Efficient Parsing for Natural Language*, volume 8 of *Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1986.

[99] G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Notices*, 26(6):107–119, 1991.

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

[101] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus schema for C/C++ preprocessing. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 75–84. IEEE Computer Society, March 2004.

[102] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 512–521, Edinburgh, United Kingdom, May 2004.

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

[104] Fuqing Yang, Hong Mei, Wanghong Yuan, Qiong Wu, and Yao Guo. Experiences in building C++ front end. *ACM SIGPLAN Notices*, 33(9):95–102, September 1998.

[105] Xiangyu Zhang and Rajiv Gupta. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 2004. To appear.

[106] 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.

[107] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, May 2003.

[108] Jianjun Zhao. Slicing concurrent Java programs. In *Proceedings of the Seventh IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 126–133, May 1999.

[109] Jianjun Zhao. A slicing-based approach to extracting reusable software architectures. In *Proc. 4th European Conference on Software Maintenance and Reengineering (CSMR'2000)*, pages 215–223, February 2000.

[110] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pages 312–320, 1996.