Függőség alapú statikus programszeletelés és közelítései

Jász Judit

Szegedi Tudományegyetem
Szoftverfejlesztés Tanszék

Témavezető: Dr. Gyimóthy Tibor
Szeged 2009. május

Értekezés doktori fokozat
megszerzéséhez

Szegedi Tudományegyetem
Informatika Doktori Iskola

Előszó

A *programszeletelés* ahhoz az eljáráshoz hasonlít, amit egy gyakorlott programozó tesz hibakereséskor annak érdekében, hogy megértse, hogy adott ponton miként viselkedik a program. Ahhoz, hogy a célját elérje, a programot kisebb részekre bontja, kiragadva azon utasításokat, amelyek valóban meghatározzák a vizsgált pont viselkedését. A kapott redukált programot nevezzük programszeletnek, amely egy önmagában is futtatható program, és amely garantálja, hogy az adott pont tekintetében ugyanúgy viselkedik, mint az eredeti program, de annál jóval egyszerűbb és kisebb. Formálisabban megfogalmazva a programszelet nem más, mint a program azon utasításainak a halmaza, amelyek potenciálisan hatnak a program egy kiemelt részén kiszámított értékre, amelyre szeletelési kritériumként hivatkozunk. A szeletelés ezen fenti definíciója Mark Weisertől származik 1979-ből [61].

Ahogy nőtt a programszeleteket meghatározó algoritmusoknak a száma, különböző módosított definícióit javasolták a szeleteknek [13; 14; 40; 58]. A definíciók közötti eltéréseknek a legfőbb oka az a tény, hogy a különböző alkalmazások a programszeletek különböző tulajdonságait követelik meg. Számos cikk, tanulmány foglalkozik a szoftverfejlesztés területén azzal, hogyan lehet definiálni, hogyan lehet módosítani a programszeleteket ahhoz, hogy újabb és újabb alkalmazásokat építhessünk rá. A programszeletelés eredeti definíciója, a mögötte álló alkalmazással, vagyis a hibakereséssel, megadja a programszeletelés témakörének egyik nagy osztályát, amelyre *hátrahaladó szeletelés*ként hivatkozunk. Hátrahaladó, mert egy adott programponttól elindulva olyan eseményeket, utasításokat keresünk a programban, amelyek végrehajtás megelőzhette az adott pontot, és amelyek okozhatták az adott pont hibás viselkedését. Persze az is lehet, hogy az alkalmazás, amely a szeletelésre épül, teljesen más motivációval bír. Amennyiben az adott programpont hatásait akarjuk vizsgálni az adott programban, akkor az *előrehaladó szeletelés* az, amellyel szűkíteni tudjuk azon utasítások halmazát, amelyet az adott pont befolyásol. Attól függően pedig, hogy egy konkrét lefutásában vizsgáljuk a programot, vagy az általános viselkedését elemezzük az adott pont tekintetében, tovább kategorizálhatjuk a szeletelést, és beszélhetünk *dinamikus* vagy *statikus szeletelés*ről.

Értekezésemben az elmúlt években végzett kutatásaim azon részeivel foglalkozom, amelyek direkt vagy indirekt módon kapcsolódnak a programszeletelés tématerületéhez,

azon belül is a statikus programszeleteléshez. Először bemutatom a bináris programok statikus szeletelése közben gyűjtött tapasztalatainkat, eredményeinket. Annak ellenére, hogy a bináris programok szeletelését is széles körben felhasználhatjuk, olyan módszert nem találtunk korábban, amely megadja a teljes interprocedurális szeletelési eljárást bináris programok esetére, és amely a közben felmerülő problémákra megoldást mutat. Ezt a hiányt próbáltuk meg pótolni az értekezés első részében bemutatott eredményekkel. A binárisok szeletelése után az értekezés második felének központi kérdése magasszintű nyelvek komponensei közötti függőségek vizsgálata. Habár a programszeletelés alkalmas eszköz lehet a legtöbb függőség felderítésében, használata nem minden esetben lehetséges, illetve hatékony. Módszert mutatunk a szeletelés által meghatározható függőségek közelítésére, amely módszert különböző gyakorlati szempontokból részletesen elemzünk.

Az értekezés mindkét része olyan gyakorlati megvalósításokat foglal össze, amelyek teljes kidolgozása egy csapatmunka eredménye. Külön köszönet a közös munkáért Kiss Ákosnak, Lehotai Gábornak, Beszédes Árpádnak, Ferenc Rudolfnak, Gergely Tamásnak, Siket Péternek, Siket Istvánnak, Sógor Zoltánnak, és természetesen témavezetőmnek, Gyimóthy Tibornak.

Ábrák jegyzéke

Táblázatok jegyzéke

Tartalomjegyzék

– Neked –

1. fejezet

Bevezető

„A legsúlyosabb hiba, ha a tények megismerése előtt kezdünk el elméleteket gyártani. Biztos, hogy a tényeket kezdjük majd el hozzáigazítani az elmélethez, pedig éppen fordítva kellene eljárni."

— *Scherlock Holmes*

Az értekezésben bemutatott tézisek központi témája, illetve eszköze a programszeletelés, bár azt, hogy pontosan mit is értünk ezen fogalom alatt, elég nehéz megválaszolni. A programszeletelést sokféleképpen definiálhatjuk attól függően, hogy mire akarjuk felhasználni, és ugyanígy sokféleképpen adhatjuk is meg.

Az értekezés bevezetőjében bemutatásra kerülnek a programszeletelés jelentősebb területei, valamint az értekezésben bemutatandó eljárások rövid összefoglalói. A bevezető feladata nem egy minden részletre kiterjedő áttekintés megadása, hiszen számos tanulmány létezik, amelyek megteszik ezt helyettünk [16; 23; 31; 36; 53; 58; 64]. Célunk csupán az, hogy az értekezésben bemutatott eredmények, módszerek, illetve azok céljai könnyebben érthetőkké váljanak.

1.1. A programszeletelés alapjai

A programszeletelés eredeti definíciója Mark Weisertől származik [61]. Az ő megfogalmazásában a programszelet a program azon utasításaiból áll, amelyek direkt, vagy indirekt hatnak a szeletelési kritériumra, ami a program egy adott utasításának, és az utasításban előforduló változók halmazának a kettőse. Weiser definíciója tulajdonképpen a *futtatható statikus hátrahaladó szeletek* egy fajtája. *Futtatható*, mivel a programszelet nemcsak utasításoknak egy lezárt halmaza, de ezek az utasítások egyben egy fordítható és futtatható programot határoznak meg. *Statikus* szeletelés, mivel meghatározásához csak

statikusan elérhető információkat használunk fel, és az eredmény független a program bármilyen bemenetétől. És végül *hátrahaladó*, mert azokat az utasításokat keressük, amik hatnak a kiválasztott utasításra, ellentétben az *előrehaladó szeleteléssel*, ahol azokat az utasításokat gyűjtjük össze, amelyekre hat a szeletelési kritérium utasítása. Weiser a programszeleteket a program vezérlési folyam gráfján megadott *adatfolyam probléma* megoldásaként határozta meg [62].

Linda és Karl Ottenstein lazítanak a programszeletek definícióján, náluk a programszelet egyszerűen azon utasítások halmaza, amelyek hatnak a kritériumban vizsgált változókra, de amelyek önmagukban nem feltétlenül alkotnak egy futtatható programot [51]. A programszeletek meghatározásához ők az úgynevezett *programfüggőségi gráf*ot veszik alapul, ahol a programszeletet egy ezen a gráfon való elérési problémaként definiálják. Ottensteinék csak *intraprocedurális szelet*eket adnak meg, vagyis egy eljárás, metódus határain kívül nem vizsgálják a programot, így az elérhető gyakorlati megvalósítások inkább Horwitz, Reps és Binkley *rendszerfüggőségi gráf*ján való megoldást veszik alapul [32], amely gráf tulajdonképpen nem más, mint Ottensteinék programfüggőségi gráfjának egy kiterjesztése.

Harman és Danicic programszelet definíciója még távolabb kerül az eredeti definíciótól. Az *amorf programszelet*nél már nem megkövetelt, hogy a szelet ugyanazokból az utasításokból álljon, mint az eredeti program, csupán az, hogy adott pontban vett viselkedése megegyezzen vele [30]. A definíciónak ez a fajta lazítására az ad okot, hogy valójában mire is szeretnénk felhasználni a programszeletelést. Az eredeti megközelítésben a szeletelést a hibakeresés egy lehetséges könnyítésének veszik, így természetes dolog, hogy az eredeti programból csak elhagyhatunk utasításokat, meg nem változtathatjuk azt. Az amorf programszeletelésnek azonban más a célja. Ezzel a technikával a szerzők a programmegértést szeretnék könnyíteni. Itt tehát nem szempont, hogy a szelet ugyanazokból az utasításokból álljon, mint az eredeti program, tényleg elég az, ha ugyanúgy viselkedik.

Nyilvánvaló, hogy egyedi definíciók egyedi megoldási technikákat követelnek meg, mégis elmondható, hogy a programszeleteléssel foglalkozó algoritmusok vagy a már megismert adatfolyam egyenleteket használják, mint Weiser algoritmusa, vagy függőségi gráfokon alapulnak, mint Ottensteinék megoldása, vagy a Bergeretti és Carre által javasolt *információ folyam*okat használják [9], amiket a program szintaktikus elemzésével kaphatunk meg.

Nemcsak a statikus, de a *dinamikus programszeletelés* megvalósításai is ezeket a megoldási lehetőségeket követik. A dinamikus programszeletelés fogalmát Korel és Laski vezették be [40], ahol a statikus szeleteléstől eltérően a szeletet a program egy adott futása alapján állítják elő. Míg a Korel-féle dinamikus szeletnél a szeleteket a megfelelő adatfolyam egyenletek megoldásával kapjuk, addig létezik a dinamikus szeleteléseknek is olyan iránya, ahol a függőségi gráfok adják a szeletek meghatározásának alapját. Agrawal

és társai adják az első függőségi gráfokon megvalósított dinamikus programszeletelési technikákat [1], ahol a függőségi gráf csomópontjait a futtatás során előállt nyomvonalak utasításai adják. A gyakorlatban azonban ez a fajta megközelítés hatalmas program reprezentációkhoz vezet.

Általában igaz az, hogy a dinamikus programszeletek sokkal pontosabb elemzést tesznek lehetővé a statikus szeletekhez képest, hiszen például egy-egy pointer értéke a dinamikus futás során egyértelműen adott, mint ahogy egy tömb műveletnél is tudjuk, hogy a tömbnek épp melyik elemét használjuk, vagy definiáljuk, sőt az is egyértelműen adott, hogy egy meghatározott esetben milyen hívási környezetből kerültünk egy végrehajtási pontra. Ez utóbbi a statikus szeletelési módszerek esetén komoly problémaforrás, hisz számtalan olyan utasítást hozhat be egy-egy szeletbe, amely a valóságban egyáltalán nem lehet hatással a kritériumra. Hátránya azonban a dinamikus programszeletelésnek az, hogy a program tetszőleges viselkedésének a leírását csak úgy tehetjük meg a segítségével, ha elegendő számú és minőségű tesztesettel futtatjuk a programot, amelyek azután igen nagy nyomvonalakat és reprezentációkat képezve igen költségessé tehetik a számításainkat.

Mivel mindkét iránynak megvan a saját hátránya, így számos megoldás és definíció keletkezett az alkalmazásoktól függően, amik a statikus és dinamikus szeletek között helyezkednek el. Canfora és társai által definiált *feltételes szeletelés* [21] a hagyományos szeletelési kritériumot egészíti ki egy, a program kezdeti állapotára tett megszorítással. Ezzel a hozzáadott feltétellel a programot egyszerűsíthetjük még azelőtt, hogy a hagyományos szeletelést elvégeznénk. Field és társai ehhez hasonló megközelítést adnak a *megszorított szeletelés*ben [26], míg Venkatesh által bevezetett *kvázi-statikus szelet*ek [60] kicsit szigorúbbak, ahol a program néhány bemenetét rögzítettnek tételezik fel, míg a maradék bemenet tetszőlegesen változhat. Amennyiben minden bemenete a programnak rögzített, úgy egy dinamikus szeletet kapunk, míg ha egy bemenet sem rögzített, akkor statikusat.

1.2. Az értekezés felépítése

Habár a programszeletelésnek ez a bevezetője közel sem teljes, arra elegendő, hogy lássuk, amikor programszeletekről beszélünk, illetve programszeleteket szeretnénk előállítani, az első legfontosabb dolog, amit meg kell határoznunk, hogy mi a célunk vele, hol szeretnénk felhasználni az eredményeket, és csak ezután kezdhetünk el foglalkozni a lényegi megvalósításokkal. Bárhogy is definiáljuk a szeletet, és bármilyen algoritmust is adunk annak meghatározására, a cél minden esetben annak a minimális szeletnek a megadása, amely kielégíti az adott szeletelési definíciót, hiszen a kapott szelet így lesz a leginformatívabb, és így fogja segíteni leginkább a rá épülő alkalmazást.

Az értekezés két nagyobb részből áll. Az első rész (2 - 5. fejezetek) bináris progra-

mok statikus szeletelésével, míg a második rész (6 - 9. fejezetek) a szeletelés egy potenciális felhasználási területével, magasszintű nyelveken írt programok függőségi analízisével foglalkozik.

Bináris programok szeletelése

Az értekezés első részében bináris programok szeletelésével foglalkozunk, megadva a bináris programok interprocedurális szeletelésének lépéseit, az egyes lépések közben felmerülő problémákat, illetve az adott problémák megoldásait. Foglalkozunk azzal, hogyan lehet a bináris program függőségi gráfjait pontosítani annak érdekében, hogy segítségével kisebb programszeleteket tudjunk előállítani. Gyakorlati mérésekkel pedig megmutatjuk az adott módszerek hatékonyságát, alkalmazhatóságát.

A 2. fejezetben összegyűjtjük a bináris programok szeletelésének irodalmába tartozó legjelentősebb eredményeket, hogy a rész további fejezeteiben bemutatott eredményeink jobban érthetőkké váljanak. A 3. fejezetben bemutatjuk, összegyűjtjük a függőségi gráfok építése során felmerülő problémákat, amelyekkel magasszintű nyelvek függőségeinek meghatározásakor nem találkozunk, és megoldást is javaslunk ezekre a problémákra. Az általunk meghatározott függőségi gráfok ezután alkalmasak lesznek arra, hogy segítségükkel meghatározzuk a bináris programok interprocedurális szeleteit. A 4. fejezetben azokat a megoldásainkat mutatjuk be, amellyel a függőségi gráfok méretét, egész pontosan a gráfokban megjelenő élek számát tudjuk csökkenteni oly módokon, hogy ezzel nem veszítünk valódi függőségeket, illetve ha veszítünk is, akkor csak olyat, ami a program adott vizsgálatában, illetve később a programszeletelésnek az adott felhasználásában nem jelent veszteséget. Végezetül a az 5. fejezetben összefoglaljuk azon gyakorlati méréseink eredményeit, amit a megelőző két fejezet módszereinek implementálásával kaptunk.

Statikus Execute After és Statikus Execute Before relációk

Az értekezés második részében magasszintű nyelvek függőségeinek vizsgálatával foglalkozunk. A fő célunk az, hogy magasszintű nyelvek programjaiban rejlő összes függőséget feltárjuk olyan hatékony eszköz segítségével, amely a ma létező nagy méretű ipari programok elemzésekor is hatékonyan használható. Azon kívül, hogy adunk egy módszert a potenciális függőségek feltérképezésére, megmutatjuk, hogy a módszerünk által adott eredmény milyen kapcsolatban áll az egyéb jelenleg használt módszerekkel, illetve a programszeletelés eredményeinek segítségével igazoljuk, hogy eljárás és osztály szinten a módszerünk nemcsak hatékony, de pontos eszköz is a függőségek meghatározására.

A második rész, hasonlóan az első részhez, egy olyan fejezettel kezdődik, a 6. fejezettel, amely egy általános képet ad arról, hogy a szakirodalomban felsorolt technikák milyen módszereket adnak magasszintű programokban rejlő függőségek meghatározá-

sára. A 7. fejezetben bevezetjük azokat a relációkat, amelyek meghatározásával próbáljuk közelíteni a programokban fellelhető függőségeket. Ezek a relációk a *Statikus Execute After (SEA)* és a *Statikus Execute Before (SEB)* relációk a program azon pontjai, komponensei között teremtenek kapcsolatot, amelyek ugyanazon útján fordulnak elő a program vezérlési folyam gráfjának. Meghatározzuk az elemzett programoknak azt a gráf reprezentációját, amely alkalmas arra, hogy segítségével megadjuk a bevezetett relációkat, illetve lehetséges algoritmusokat is adunk ezek megadására. Bár a programszeletelés alkalmas lehet arra, hogy a programban rejlő függőségeket pontosan közelítse, nagy szoftverrendszerek esetén a szelet meghatározása nem hatékony. Annak érdekében, hogy lássuk, az általunk adott módszer hatékony és pontos eszköz is lehet a program megfelelő szintjén lévő elemek közötti függőségek vizsgálatakor, gyakorlati mérésekben összehasonlítottuk a CodeSurfer [29] programszeletelő eszköz által adott szeletelési eredményeket és az általuk meghatározott SEA, illetve SEB relációkat. Ezeket a gyakorlati összehasonlításokat tartalmazza a 8. fejezet. Köszönhetően a CodeSurfer általunk használt verziójában (2.1p1) talált pontatlanságnak, a gyakorlati mérések egy része csak a hátrahaladó irányban volt kivitelezhető, vagyis a hátrahaladó szeletek és SEB relációk között. (Az, hogy mi volt ez a pontatlanság, és hogyan befolyásolta a mérési eredményeinket, a függelék A részében mutatjuk be.) A 9. fejezetben bemutatjuk, hogy objektumorientált nyelvek osztály szintű kapcsolatainak meghatározásához hogyan használhatjuk a SEA és SEB relációkat a megfelelő program transzformáció végrehajtása után. Gyakorlati méréseinkben pedig arra keressük a választ, hogy az átlagos technikák, metrikákon alapuló módszerek, amelyeket számos esetben használnak az objektumorientált programok osztályai közötti kapcsolatok megadására, mennyire lehetnek megbízhatóak.

Az értekezés eredményeit a 10. fejezetben értékeljük, illetve a függelék B és C részeiben adott magyar és angol nyelvű összefoglalókkal zárjuk.

1.3. Az értekezés eredményei

Az értekezésben ismertetett eredményeket az eddigiek alapján két nagyobb csoportba tehetjük. Az alábbiakban az egyes csoportokon belül kiemeljük, illetve tézisekbe foglaljuk az elért eredményeket.

Bináris programok szeletelése

Az értekezés első felében bináris programok szeletelésével foglalkozunk. Célunk egy olyan szeletelési eljárás megadása, amely a szeletelés eredeti definíciójának megfelelően képes bináris programok interprocedurális szeleteit meghatározni. Mivel bináris programok szeletelésének hasonló felhasználási területei lehetnek, mint a magasszintű nyelvek szeletelésének, sőt még további felhasználási lehetőségek is adódhatnak, így fontosnak találtuk,

hogy bináris programokon is definiáljuk a programszeletelés fő lépéseit, megadjuk azokat a problémákat, amelyekkel binárisok szeletelésekor találkozhatunk, illetve megmutassuk, hogy a gyakorlatban alkalmazva a bevezetett technikákat milyen eredményeket értünk el. A magasszintű nyelveknél megismert területeken kívül bináris programok szeletelésének felhasználási területei lehetnek a rossz szándékú bináris programok, vagy vírusok, férgek működésének megértése, vagy biztonsági rések kiszűrésének megkönnyítése.

Bár az értekezésben végig többes számot használok a megvalósításaink bemutatásakor, az értekezés tézisei mind olyan pontjait foglalják össze kutatásainknak, amelyeket részben, vagy teljesen a saját eredményeimnek tudhatok. A tézispontok megfogalmazásakor, illetve a megfelelő fejezetek végén kiemelem, hogy az abban bemutatott eredmények mely része saját eredmény, és melyek a szerzőtársakkal közös, oszthatatlan eredmények. A bináris programok szeleteléséhez tartozó eredményeinket a [38; 39] cikkek alapján a következő tézispontokban foglalom össze.

I/1 **Bináris programok függőségi gráfjainak előállítási problémái és megoldásai, függőség alapú szeletelés adaptálása bináris programokra.**
Habár számos felhasználási területe lehet a bináris programok szeletelésének, a szakirodalom ezzel a témával csak részlegesen foglalkozik. Az egyik legelterjedtebb szeletelési eljárás az, amikor a szeletelést a program függőségi gráfjain valósítják meg. Ahhoz, hogy a függőségi gráfokat meghatározhassuk bináris programok esetében, számos olyan problémára kell megoldást találnunk, amik magasszintű nyelveknél nem fordulnak elő. Ahhoz, hogy megadhassuk bináris programok interprocedurális szeleteit, összegyűjtöttük a szeleteléshez szükséges függőségi gráfok meghatározásakor felmerülő problémákat, és ezen problémákra megoldásokat adtunk. Ezek közül a vezérlésfüggőségi, adatfüggőségi és a rendszerfüggőségi gráfok megadása saját eredményem. A bináris programok függőségi gráfjainak építésével a 3. fejezet foglalkozik.

I/2 **Módosított szeletelési algoritmusok kidolgozása a bináris programok szeletelésének pontosítására.**
A statikus szeletelés egyik legnagyobb hátrányának tudják be azt a tényt, hogy az esetek többségében a számított szeletek olyan nagyok, hogy azzal nem tudják gyakorlatilag hatékonyan javítani a szeletelésre épülő alkalmazás munkáját. Az, hogy a szeletméretek igen nagyok lehetnek annak köszönhető, hogy a statikus elemzés során minden potenciálisan létrejövő függőséget kezelnünk kell ahhoz, hogy a szeletelés eredménye biztonságos maradjon. Ez azt jelenti, ha valami miatt a program két pontjáról nem tudjuk eldönteni egyértelműen, hogy azok kapcsolatban állnak-e egymással, vagy sem, akkor úgy kell tekintenünk, mintha az előbbi eset teljesülne. Ennek következtében a statikus szeletelési eljárás elég konzervatív lehet, ami végeredményében a nagy szelet méretekhez vezethet.

A függőségi gráfok pontosítására két statikus és két dinamikus módszert vezettünk be. Saját eredményem a statikus pontosításoknál a szerzőtársam által bevezetett adatfüggőségi háló révén megvalósított pontosított adatfüggőségi elemzés, míg a többi módszer megadása a szerzőtársakkal közös eredmény. Az értekezés 4. fejezete foglalja össze ezt a témát.

I/3 **Bináris programok statikus szeletelésének gyakorlati kiértékelése.**
Az előző két tézispontban megfogalmazott eljárásokat gyakorlatban is megvalósítottuk, illetve kiértékeltük statikusan linkelt ARM programokon. Célunk a méréssel nemcsak az, hogy megmutassuk az egyes technikák hatékonyságát, de az is, hogy a statikus programszeletelés használhatóságát igazoljuk. A mérések megtervezése és kiértékelése a szerzőtársakkal közös eredmény, míg a vezérlési folyam gráfon kívül a függőségi gráfok gyakorlatban való megadása, illetve az ismertetett módszerek implementálása saját eredmény. A gyakorlatban kapott eredményeinket az 5. fejezet tartalmazza.

Statikus Execute After és Statikus Execute Before relációk

Az értekezés második részének témája a program egyes komponensei között létrejövő függőségek vizsgálata. Egy adott programban rejlő függőségeknek a feltárása igen fontos feladata a szoftverfejlesztés számos ágának. A programszeletelés az eredeti formájában ezen függőségek feltérképezésében is használható lehet, bár elképzelhető, hogy a program különböző szintű reprezentációin más és más módszerek használata lehet kedvezőbb. Az értekezésnek ebben a második részében bevezetünk egy reláció párost, amely alkalmas arra, hogy a programszeletelés eredményeit hatékonyan helyettesítse bizonyos esetekben. Ezen a területen végzett kutatásainkat a [12; 33; 34] cikkekben publikáltuk, az eredményeket pedig a következő tézispontokban foglalom össze.

II/1 **Statikus Execute After (SEA) és Statikus Execute Before (SEB) relációk definiálása, és program reprezentáció kidolgozása ezen relációk meghatározásához. Algoritmusok megadása a SEA és SEB relációk meghatározására.**
Nagy szoftverrendszerek esetében, amelyek kódmérete több milliós is lehet, a függőség alapú programszeletek meghatározása szinte lehetetlen a nagy program reprezentáció miatt. Szoftverfejlesztés során a program komponensei között lévő függőségek ismerete elengedhetetlen ahhoz, hogy egy megbízhatóan működő programot fejlesszünk. Felhasználva Apiwattanapong és társai által definiált *Execute After* relációkat [3], amelyeket a program futtatásaival határozhatunk meg dinamikusan, definiáltuk a program eljárásai között létrejövő Statikus Execute After és Statikus Execute Before relációkat. Ezen relációk segítségével megpróbáltuk

közelíteni a programban statikusan megjelenő függőségeket. Mivel ezeknek a relációknak a meghatározása sokkal egyszerűbb, mint a programszeletelés, még nagy rendszerek esetén is hatékonyan használhatóak. Ahhoz, hogy ezeket a relációkat megadjuk, egy erre alkalmas programreprezentációt definiáltunk, az *Interprocedural Component Control Flow Graph*-ot *(ICCFG)*. A SEA és SEB relációk a bevezetett program reprezentáció segítségével viszonylag könnyen meghatározhatóak. Adott program esetében elképzelhető, hogy nem probléma az összes reláció memóriában tartása egy időben, de elképzelhető az is, hogy erre nincs lehetőségünk. A megadott algoritmusaink a végleteket emelik ki, azaz azt az esetet, amikor minden relációt egyidőben számítunk ki, és minden relációt egyszerre tárolunk a memóriában, illetve azt az esetet, amikor a meghatározott relációkat nem tároljuk, csupán a megfelelő program reprezentáció áll a rendelkezésünkre, és csak megadott pontok relációit akarjuk meghatározni. A SEA és SEB relációk, valamint az ICCFG gráf definiálása a szerzőtársakkal közös eredményem, míg a bemutatott algoritmusok és azoknak a vizsgálata saját eredményem. A 7. fejezet részletezi ezt a pontot.

II/2 **A SEA és SEB relációk gyakorlati összevetése a programszeleteléssel.** Ahhoz, hogy belássuk, hogy a SEA és a SEB relációk valóban alkalmasak arra, hogy a programban rejlő függőségeket hatékonyan közelítsék eljárás, illetve annál magasabb szinten, az algoritmusainkat implementáltuk, és a kapott eredményt összevetettük különböző programszeletelő eszközök eredményeivel. Természetesen attól függően, hogy a programszeletet hogyan definiáljuk, és az alkalmazott szeletelő eszköz milyen pontossággal határozza meg a programban rejlő függőségeket, illetve hogy a szeletelő eszköz valóban megtalálja-e az összes függőséget, az összehasonlításunk eredménye eltérő lehet a különböző eszközök esetében. Ennek ellenére mondhatjuk, hogy megfelelő mennyiségű teszt által reális képet kaphatunk arról, hogy a bevezetett relációk mennyire használhatóak a programban lévő függőségek közelítésére. A [12] cikkben C++ és Java programokban, míg a [33; 34] cikkekben C/C++ programokban meghatározható SEA, illetve SEB relációkat vetettük össze a szeletelő eszközök eredményeivel. A gyakorlati mérések megtervezése és kivitelezése saját eredményeim. Az értekezésben csak a C/C++ programokon meghatározott eredmények kerülnek ismertetésre a 8. fejezetben.

II/3 **Rejtett függőségek objektumorientált programokban; a SEA és SEB relációk gyakorlati vizsgálata objektumorientált programokban.** Objektumorientált programok hatásanalízise során számos eljárás épít arra, hogy az egyes osztályok között expliciten észrevehető kapcsolatok alapján próbálja meghatározni egy potenciális változtatás hatását. Habár ezek a módszerek mások szerint is alkalmatlanok arra, hogy a programban létrejövő valamennyi függőséget

feltérképezzék, és csupán a legvalószínűbb függőségeket adják vissza, egyszerűségük miatt mégis elterjedtek. A 9. fejezetben gyakorlati példákkal igazoljuk, hogy az objektumorientált programokban expliciten megjelenő függőségek valóban nem tárják fel az összes függőséget, és így ezek önmagukban csak arra képesek, hogy a nagy valószínűséggel létrejövő függőségeket azonosítsák be, de az olyan alkalmazásokban, ahol valamennyi függőség meghatározása szükséges, ezek nem biztonságos eszközök. A mérések megtervezése a szerzőtársaimmal közös eredménynek tekintendő, míg a C/C++ programokhoz kapcsolódó mérések kivitelezése, valamint a SEA és SEB relációk gyakorlati megadása a saját eredményeim.

Az áttekinthetőség érdekében a tézisek és a kapcsolódó publikációk kapcsolatát a következő táblázatban is összefoglaljuk:

	[39]	[38]	[12]	[34]	[33]
I/1	•				
I/2	•	•			
I/3	•	•			
II/1			•	•	•
II/2			•	•	•
II/3			•		

I. rész

Bináris programok szeletelése

2. fejezet

Bevezetés bináris programok szeletelésébe

„Csak 10-féle ember él a világon: vannak azok, akik tisztában vannak a bináris jelentésével, és vannak, akik nem."

— Ismeretlen szerző

Számtalan cikk foglalkozik a magasszintű programozási nyelvek szeletelésével, de viszonylag kevés figyelmet kaptak a bináris programok ebben a témakörben annak ellenére, hogy bináris programok szeletelésének felhasználási területei túlmutatnak a magasszintű nyelveknél megismert területeken. Bináris programok szeletelését felhasználhatjuk arra, hogy megértsük olyan kódok viselkedését, amelyeknél valami okból nem áll rendelkezésünkre annak a forrása, akár mert az adott program egy örökölt kód, amelyet már nem tartanak karban, vagy egy vírus, vagy csak egy olyan bináris, amelyet linkelés után módosítottak valamilyen módon, de nyilvánvaló felhasználhatjuk azokra a dolgokra is, amit a magasszintű nyelveknél megismertünk.

Bár nem találkoztunk a szakirodalomban olyan munkával, amely megadja a teljes interprocedurális szeletelési eljárás lépéseit, problémáit bináris szinten, a szeletelés részfeladatainak megoldására találunk törekvéseket. Ahhoz, hogy a bináris programok szeleteit meg tudjuk határozni, természetesen szükségünk van a bináris adatokból meghatározható vezérlési folyam információkra. Debray és társai a kódtömörítési eljárásukhoz [24] adnak egy módszert arra, hogyan tudják felépíteni a vezérlési folyam gráfját Alpha architektúrára fordított binárisoknak. Kastnerék célja egy általános, azaz gépfüggetlen vezérlési folyam gráf megadása [37]. Mivel a beágyazott rendszerek processzorai sokszor eltérnek a szabványos hardware architektúráktól, ennek következtében a kódgenerálás és optimalizációs technikák elbukhatnak. Módszerük segítségével azonban lehetőség nyílik arra,

hogy tetszőleges binárisból a lehető legoptimálisabb kódot állítsák elő adott architektúrára. Larus és Schnarr intraprocedurális statikus szeletelést alkalmaznak az EEL-nek keresztelt bináris szerkesztő eszközükben [42]. A szeletelést arra használják fel, hogy pontosítsák a vezérlési folyam analízisüket az indirekt hívásoknál, amelyek leggyakrabban a `switch` utasítások fordításából származnak. Hátrahaladó szeletelés segítségével képesek arra, hogy architektúra- és fordító független módon elemezzék ezeket a helyzeteket. Cifuentes és Frabuolet szintén adnak egy intraprocedurális eljárást bináris programok szeletelésére [22]. Bemutatják, hogy a hagyományos intraprocedurális szeletelési technikákat hogyan lehet alkalmazni binárisok esetében arra, hogy gépi és assembly kódokat elemezzenek, hogy ezekben a kódokban hibát keressenek, illetve hogy meghatározzák azokat az utasításokat, amelyek indexelt ugrást befolyásolnak, illetve amelyek hatnak arra a regiszter értékre, amely egy indirekt hívás helyét jelöli ki. Bergeron és társai az elsők, akik ugyan javasolják, hogy rosszindulatú kódok viselkedésének feltérképezéséhez bináris programok interprocedurális szeletelését kellene alkalmazni, és hogy ezt függőségi gráfokkal lehetne megadni [10], de gyakorlati megoldásokat nem nyújtanak ők sem a binárisok interprocedurális szeletelése közben felmerülő problémák kezelésére.

Balakrishnan és Reps a mi kutatásainkkal egy időben x86-os binárisok statikus elemzését vették célba [6]. Az ő törekvésük és céljuk nagyon hasonló a miénkhez, bár megvalósításban eltér attól. Az ő fő céljuk az, hogy a magasszintű nyelveknél megismert köztes programreprezentációkat használják binárisok statikus elemzéshez oly módon, hogy ezek előállításakor ne kelljen nekik felhasználni a szimbólum táblákban rejlő információkat, és ne kelljen használniuk egyéb nyomkövető információkat. Az egyetlen forrás, amelyből információkat nyernek az az, hogy feltérképezik a memória lehetséges tartalmát, illetve figyelik, hogy az elemzett bináris azt hogyan használhatja, illetve módosíthatja.

Az értekezés jelen részében a bináris programok interprocedurális statikus szeletelésének témakörében gyűjtött tapasztalatainkat, eredményeinket foglaljuk össze. Fő motivációnk a fentieknek megfelelően, hogy bár az alkalmazási területei a statikus programok interprocedurális szeletelésének igen széleskörű, ennek előállításával, és a közben felmerülő problémákkal mások nem foglalkoztak előttünk. Kutatásaink másik alapját az adta, hogy a statikus szeletelések eredményeit sokan használhatatlannak kezelik, lévén sok esetben a szelet méretek túl nagyok a program méretéhez viszonyítva, aminek a következménye, hogy az alkalmazás, amelyet az eredményeire építünk, nem lesz hatékonyabb. Célunk, hogy megmutassuk, a gyakorlati életben nagyobb bináris programok esetében is jelentős méretbeli különbség lehet a szelet méretek és a program méret között. Végül megmutatjuk, hogy dinamikus adatokkal pontosítva tovább redukálhatóak a statikusan meghatározható szelet méretek. Ez a szeletelési technika hatékonyan használható az olyan alkalmazásoknál, amelyek nem követelik meg, hogy a meghatározott programszelet mindenképp felfedje az adott programpont összes függőségét. Ilyen lehet például a

hibakeresés is, ahol már ez a jóval pontatlanabb eszköz is segítség lehet a hiba gyors megtalálásában.

3. fejezet

Bináris programok függőségi gráfjainak meghatározása

„Aki a célt akarja, annak az eszközöket is akarnia kell."

— Stendhal

Az értekezésben bemutatandó eredmények első nagy részében tehát bináris programok interprocedurális szeletelésének folyamatát adjuk meg. Ehhez először ismertetjük azon problémákat, amelyekkel akkor találkozhatunk, amikor a bináris program függőségi gráfjait próbáljuk felépíteni. Megoldást mutatunk ezen problémákra, és részletesen bemutatjuk az egyes függőségi gráfok előállításának lépéseit. A bináris programoknak a megadott függőségi gráfokon alapuló reprezentációja alkalmas lesz arra, hogy azon programszeleteket állítsunk elő.

A függőségi gráfok meghatározása során először meg kell határoznunk a program vezérlési folyam gráfját *(Control Flow Graph - CFG)*, majd az adatfüggőségi gráf *(Data Dependence Graph - DDG)* és vezérlésfüggőségi gráf *(Control Dependence Graph - CDG)* együttesen meghatározzák valamennyi eljárásnak a programfüggőségi gráfját *(Program Dependence Graph - PDG)*, amelyeket a megfelelő élekkel kiegészítve megkapjuk a rendszerfüggőségi gráfot *(System Dependence Graph - SDG)*, amelyet közvetlenül felhasználunk az interprocedurális szeletek kiszámításához.

3.1. Vezérlési folyam analízis

Általában igaz az, hogy magasszintű nyelveknél a program szemantikáját reprezentáló vezérlési folyam gráf, vagy röviden vezérlési gráf, megadása viszonylag egyszerű feladat. Bináris programok esetében azonban számtalan olyan probléma jelentkezik a szeletelésnek már ebben az első lépésben is, amivel a magasszintű nyelveknél nem találkozunk.

A bináris program nem más, mint egy magasszintű, vagy assembly program gépi kódú megfelelője, amelyet minden egyes platformra (gép és operációs rendszer) külön fordítunk és linkelünk. Bináris program esetében a program bájtok sorozatából áll. Ahhoz, hogy a program vezérlését elemezhessük, a programot újra vissza kell állítanunk a bináris változatából. A program alacsonyszintű utasításainak behatárolásával kapjuk a vezérlési gráf csomópontjai. Az egyes utasításokra jellemző viselkedés alapján határozhatjuk meg a vezérlési folyam gráf éleit, a vezérlési, hívási és visszatérési éleket. Ez a feladat nehezebb, mint magasszintű nyelveknél, hiszen:

- a bináris utasítások száma messze meghaladhatja a magasszintű nyelvek vezérlési szerkezeteinek a számát,
- egy bináris program több utasításkészletet használhat egy időben[1],
- változó hosszú utasítások lehetnek,
- függvényhatárok elmosódhatnak köszönhetően az átlapolásoknak, keresztugrásoknak,
- a hívási gráf pontos megadása is nehézkes lehet a nagy számú indirekt hívásoknak köszönhetően[2],
- nemcsak a függvények között, hanem egy függvényen belül is elképzelhető olyan ugrás, amelynek a célpontja statikusan nem meghatározható.

A fenti felsorolás rávilágít arra, hogy a bináris programok vezérlésének elemzése számos nehézséget rejt magában. Általános megoldást nehéz adni a felsorolt problémákra, de néhány segédinformáció, illetve szerkezeti heurisztika segítheti ezek lekezelését.

A legtöbb fájl formátum, ami magában hordozza a program bináris képét, tartalmazza azon extra információkat is a nyers bináris adatok mellett, amelyekre szükségünk lehet a program vezérlési folyam gráfjának előállításakor [47; 59]. A fordítók, assemblerek és linkerek általában szimbolikus és relokációs információkat tárolnak el a legyártott fájlokban. A szimbolikus információk felhasználhatóak arra, hogy elkülönítsük a kód és adat részeket a program bináris képében, vagy segíthetnek a függvények határainak megtalálásában és az utasításkészlet kiválasztásában. A relokációs információk pedig segíthetnek az indirekt függvényhívások feloldásában és a nem egyértelmű vezérlés átadások esetében. Általában a kézzel írt assembly kód szintén elemezhető, bár elképzelhető, hogy némi felhasználói segítségre szükség van ehhez.

[1] A 5. fejezet gyakorlati méréseiben mi olyan bináris programokat vizsgálunk, ahol a 32-bites ARM kód keveredik a 16-bites Thumb kódokkal.

[2] Ezzel a problémával magasszintű nyelveknél is találkozhatunk, de mégis ez inkább az alacsonyszintű programokra jellemző. Persze ennek oka az is, hogy magasszintű nyelveknél az egyes függvények szignatúrája is segíthet a hívás céljának beazonosításában, míg bináris programok esetében erre nincs lehetőség.

Nyilvánvaló, hogy az eltárolt információk erősen gép és operációs rendszer függőek, így nem lehet általánosítani azt, hogyan lehet használható adatokat kinyerni ezekből. Hasonlóan arra sincs általános megoldás, hogy a különböző platformokon futó utasítások viselkedését miként lehetne általánosan elemezni. Habár ezek újabb problémáknak tűnnek, a gyakorlat azt mutatja, hogy ezek leküzdhetők megfelelő fordító, fájl formátum és architektúra specifikációk használatával.

Tegyük fel, hogy adott bináris esetében legyőzve a fenti problémákat, nekiláthatunk a függőségi gráfok felépítésének. Ahhoz, hogy a későbbi gráf bejárásaink hatékonyabbak legyenek, az utasítások elemzésével meg kell határoznunk az alapblokkokat, amelyek utasítások olyan csoportjai, amelyekre igaz az, hogy a bennük lévő utasításokat mindig egymás után hajtjuk végre, az elsőtől az utolsóig. Az alapblokkok meghatározása után ezek további csoportosításával pedig meg kell határoznunk a függvényeket.

Az alapblokkok meghatározásához meg kell keresnünk az egyes blokkok belépési pontjait, kezdő utasításait. Ilyenek azok az utasítások, melyek függvényhívás, vagy ugrás után helyezkednek el, illetve amelyekre egy ugró utasítás ugorhat, továbbá a függvények első, valamint az utasításkészletet váltó utasításokat követő utasítások. Az így meghatározott utasítások által közrefogott utasítások alkotják a függvények alapblokkjait, amelyeket a megfelelő vezérlési élekkel kötünk össze a függvényen belül. Minden függvényt kiegészítünk egy további blokkal, az **exit** blokkal. Ez reprezentálja a függvény egyetlen visszatérési pontját. Ezt úgy érjük el, hogy valamennyi alapblokkot, amelynek végén a függvény visszatérhet, egy vezérlési éllel összekötjük ezzel az **exit** blokkal.

A függvényhívást kezdeményező alapblokkot *hívási pont*nak, ahová egy függvényhívás után visszatérünk *visszatérési pont*nak nevezzük. A hívási pontokat és az általuk meghívott függvény csomópontokat egy-egy hívó éllel, míg a függvények **exit** csomópontjait és a visszatérési pontokat egy-egy visszatérési éllel kötjük össze a vezérlési gráfban.

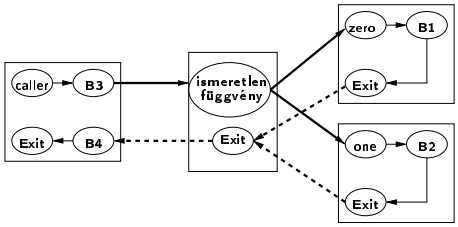
Speciális esettel állunk szemben, ha olyan ugró utasítással vagy függvényhívással találkozunk, melynél nem tudjuk meghatározni a vezérlésátadás célját. Ilyen szerkezetek tipikusan a C programok `switch` utasításainak fordításakor, valamint függvény pointerek használatakor jelentkeznek. Ezen esetek kezeléséhez a CFG-t speciális csomópontokkal egészítjük ki. A fel nem oldott függvényhívások céljait reprezentáló csomópontot **ismeretlen függvény**nek nevezzük. Ide kötjük be azon hívási éleket, amelyek céljait nem tudjuk egyértelműen meghatározni. A bevezetett **ismeretlen függvény** csomópontra úgy tekintünk, mintha az egy olyan függvényt reprezentálna, amiből valamennyi olyan függvényt hívhatnánk, amelynek valahol a program folyamán a címét meghatározták, hiszen ez szükséges feltétele annak, hogy arra indirekt hívást kezdeményezhessen valaki. **Ismeretlen függvény** csomópont bevezetésére mutat példát az 3.1. ábra.

Hasonlóan ehhez a csomóponthoz minden függvénynek lesz egy **ismeretlen blokk** csomópontja, amelybe vezérlési élek vezetnek a függvény fel nem oldott ugrásainak

Kód

```
zero:   MOV R0,#0              ;B1
        MOV PC,LR

one:    MOV R0,#1              ;B2
        MOV PC,LR

caller: STMFD R13!,{R4,LR}     ;B3
        MOV R4,R0
        MOV R14,PC
        MOV PC,R4 ;indirekt hívás
        LDMFD R13!,{R4,PC}     ;B4
```

CFG

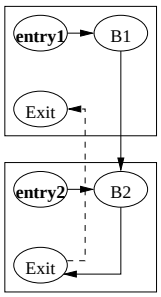


3.1. ábra. Indirekt függvényhívás két lehetséges célponttal ARM kódon. A CFG vékony élei jelölik a vezérlési éleket, a vastag élek a hívási élek, míg a szaggatott élek a visszatérési élek. Az alapblokkok a bal oldali kód jelölésének megfelelőek. Minden függvény **entry** csomópontját az adott függvény nevével azonosítjuk.

Kód

```
entry1: STR R0,[R13,#4]    ;B1
        ;vezérlés "átfolyása"

entry2: LDR R0,[R13,#8]    ;B2
        SUB R0,#1
        STR R0,[R13,#8]
        MOV PC,r14
```

CFG



3.2. ábra. Átlapoló függvények ARM kódon. A CFG-ben a vékony élek a vezérlési élek, míg a szaggatott él a kompenzációs vezérlési él.

blokkjairól, és amelyből vezérlési élek indulnak az összes lehetséges blokkhoz, amelyek céljai lehetnek az előző ugrásoknak.

Az átlapolásokat és a keresztugrásokat megvalósító függvények szintén speciális elbánást igényelnek. Ha egy függvényre a vezérlés úgy kerül egy másik függvényből, hogy ott nem történt függvényhívás, akkor annak érdekében, hogy kompenzáljuk a visszatérési él hiányát, a hívott függvény **exit** blokkját össze kell kötnünk a hívó függvény **exit** blokkjával. Az ilyen átlapolások és keresztugrások általában az erőteljes fordító optimalizálásoknak köszönhetőek, illetve azokban a nyelvekben gyakoriak, ahol egy függvénynek több belépési pontja lehet. A 3.2. ábra ARM kód mutatja be az átlapolást, illetve a neki megfelelő CFG reprezentációt.

Ahhoz, hogy a későbbiekben szemléltetni tudjuk a megfelelő függőségi gráfok felépítésének lépéseit, és a bináris programszeleteinek előállítását, tekintsük a 3.3. ábra

```
          add:
00002ECC  ADD R0, R1, R0          B1
00002ECE  MOV PC, LR
          mul:
00002ED0  PUSH {R4,R5,LR}         B2
00002ED2  ADD R4, R1, #0
00002ED4  ADD R5, R0, #0
00002ED6  MOV R3, #0
00002ED8  MOV R2, #1
00002EDA  CMP R2, R4
00002EDC  BGT 0x00002EF6
00002EDE  ADD R0, R3, #0          B3
00002EE0  ADD R1, R5, #0
00002EE2  BL 0x00002ECC (add)
00002EE6  ADD R3, R0, #0          B4
00002EE8  ADD R0, R2, #0
00002EEA  MOV R1, #1
00002EEC  BL 0x00002ECC (add)
00002EF0  ADD R2, R0, #0          B5
00002EF2  CMP R2, R4
00002EF4  BLE 0x00002EDE
00002EF6  ADD R0, R3, #0          B6
00002EF8  POP {R4,R5,PC}
          main:
00002EFA  PUSH {R4,R5,LR}         B7
00002EFC  ADD SP, #-8
00002EFE  BL 0x00002EAC (readin)
00002F02  ADD R5, R0, #0          B8
00002F04  MOV R0, #0
00002F06  STR R0, [SP, #0]
00002F08  MOV R0, #1
00002F0A  STR R0, [SP, #4]
00002F0C  MOV R4, #1
00002F0E  CMP R4, R5
00002F10  BGE 0x00002F2A
00002F12  LDR R0, [SP, #0]        B9
00002F14  ADD R1, R4, #0
00002F16  BL 0x00002ECC (add)
00002F1A  STR R0, [SP, #0]        B10
00002F1C  LDR R0, [SP, #4]
00002F1E  BL 0x00002ED0 (mul)
00002F22  STR R0, [SP, #4]        B11
00002F24  ADD R4, #1
00002F26  CMP R4, R5
00002F28  BLT 0x00002F12
00002F2A  LDR R0, [SP, #0]        B12
00002F2C  LDR R1, [SP, #4]
00002F2E  BL 0x00002EBE (writeout)
00002F32  ADD SP, #8              B13
00002F34  POP {R4,R5,PC}
```

3.3. ábra. Diszasszemblált bináris program kivonata. Az alapblokkok vízszintes vonalakkal vannak elválasztva egymástól.

példáját, ahol egy ARM architektúrára fordított program diszasszemblált kódja látható. A program kiszámítja az első N szám összegét és szorzatát, majd kiírja az eredményeket. Az 3.4. ábra ennek a példának a CFG-jét szemlélteti.

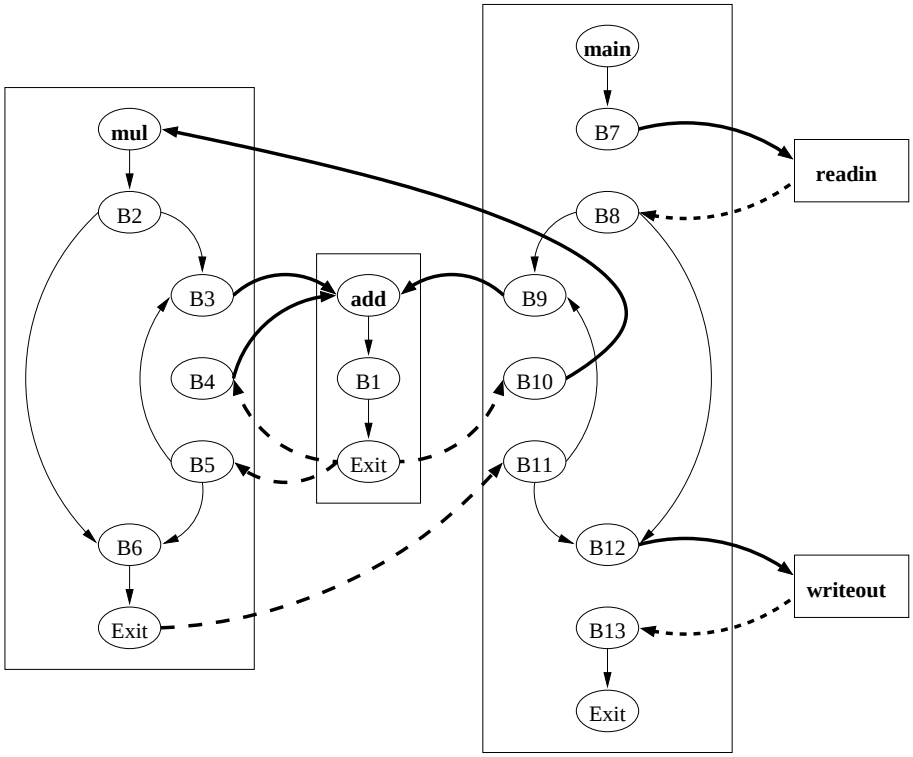
3.2. Programfüggőségi gráf meghatározása

A programfüggőségi gráf felépítése két nagyobb lépésből áll. Először meg kell határoznunk a vezérlés függéseket egy-egy függvényen belül, utána pedig meg kell adnunk az adat függéseket az egyes csomópontok között.

3.2.1. Vezérlés függőségek meghatározása

Ahhoz, hogy a vezérlés függéseket megadhassuk, az egyes alapblokkok közötti posztdominancia relációkat[3] kell összegyűjtenünk. Ehhez használhatjuk Lengauer és Tarjan klasszikus algoritmusát [46], amely logaritmikus időben képes meghatározni egy gráf csúcspontjainak a közvetlen dominátorait. Megjegyezzük, hogy a (poszt)dominátor számításra adtak lineáris idejű algoritmusokat is [2; 20; 28], amelyek azonban vagy

[3]Az M blokk posztdominálja az N blokkot, ha minden N-ből az **exit** csomópontba tartó út tartalmazza M-et.



3.4. ábra. Az 3.3. ábrán megadott program CFG-je. A vékony élek a vezérlési, a vastag élek a hívási, a szaggatott élek a visszatérési éleket jelölik. Az alapblokkok megfelelnek az 3.3. ábra jelölésének.

sokkal összetettebbek, vagy némely esetben pontatlannak bizonyultak. Mivel az egy függvényhez tartozó alapblokkok száma és az ezekhez kapcsolódó élek száma a teljes rendszerfüggőségi gráf csomópontjainak és éleinek átlagosan csak a töredékei[4], így a teljes szeletelés összköltségéhez képest nem mérvadó, ha a logaritmikus időköltségű megoldást választjuk.

Az átlapolásoknak köszönhetően bináris programoknál előfordulnak olyan esetek, amikor egy alapblokkba más függvényből is indul vezérlési él, nemcsak abból, amelyikbe beletartozik. A posztdominátorokat függvényenként számoljuk, bár az átlapolásoknak és keresztugrásoknak köszönhetően ezt egy kicsit sajátságosan kell értelmeznünk. Az éppen vizsgált függvény **entry** csomópontjából kiindulva a vezérlési éleken haladva gyűjtsük össze az összes alapblokkot, míg a függvény **exit** blokkját el nem érjük! A függvény csomópontjait tehát nem feltétlenül azok a csomópontok alkotják, amelyek magában a függvényben előfordulnak. Ráadásul előfordulhat olyan eset, hogy egy blokkra nem kerül rá a vezérlés. Ekkor erre a blokkra nem is számítunk posztdominátort, hisz nem kell azzal sem törődnünk így, hogy más blokkok végrehajtása függ-e ettől a blokktól.

[4] Az 5. fejezetben ismertetett gyakorlati méréseinknél az átlagosan egy függvényre jutó alapblokkok száma kevesebb volt 30-nál a legnagyobb program esetében is.

Találkozhatunk például olyan esettel is, amikor egy függvény szinte „bekebelezi" egy másik függvény blokkjait. Ilyenkor a másik függvény **entry** csomópontjából nem indul, illetve az **exit** csomópontjába nem megy vezérlési él. Szintén gyakori, amikor egy függvény „ráfolyik" egy másik függvényre. Ez akkor fordulhat elő, amikor a két függvénynek csak az eleje különbözik, a végeik azonban megegyeznek, ahogy azt a 3.2. ábra példája is mutatja. Ilyenkor azon függvény **exit** csomópontjából, melyre „ráfolyt" a másik, a CFG felépítésének köszönhetően vezérlési él megy az eredeti függvény **exit** csomópontjába. Amikor az első függvényünket vizsgáljuk, és elindulunk annak **entry** csomópontjából, hogy összegyűjtsük az érintett blokkokat, bár „elkalandozunk" más függvény blokkjaihoz, mégis vissza tudunk térni az eredeti függvény **exit** csomópontjába. Nyilvánvalóan ebben az esetben a posztdominátor számítás után végzett vezérlés függések meghatározása során azt kaphatjuk, hogy egy blokk több függvény blokkjaitól áll vezérlés függésben. Statikus esetben természetesen nem tudjuk eldönteni, hogy éppen melyik függés van életben, így azt kell feltételeznünk, mintha az összes függés előfordulhatna.

A posztdominátorok ismeretében a vezérlés függések meghatározása már egyszerű feladat [49]. Az éppen vizsgált függvény CFG^+ gráfja legyen az a gráf, amelyet az adott függvény CFG-jéből kapunk azzal a kiegészítéssel, hogy az **entry** és **exit** csomópontok között is adott egy él! Ezután ezen a CFG^+ gráfon meghatározzuk a posztdominancia relációkat. Legyen S a CFG^+ azon (m, n) éleinek a halmaza, amelyeknél n nem posztdominálja m-et! Ezután minden $(m, n) \in S$-beli élre határozzuk meg a legkisebb közös ősét m-nek és n-nek a posztdominancia fában! Legyen ez a csomópont l, ami vagy az m, vagy pedig annak valamely őse! Ekkor a posztdominancia fában az (l, n) út valamennyi csomópontjára az l csomóponton kívül igaz az, hogy vezérlés függésben áll az m csomóponttól. Adódik, hogyha egy alapblokkra, vagyis egy csomópontra igaz az, hogy az a függvény **entry** csomópontjától áll vezérlés függésben, akkor arra úgy tekintsünk, mintha a függvényt hívó blokktól állna vezérlés függésben.

3.2.2. Adat függőségek meghatározása

A szeletelés folyamatának soron következő feladata az adat függőségek feltárása, azaz az adatfüggőségi gráf meghatározása. Magasszintű nyelveknél az utasítások argumentumai általában lokális vagy globális változók, esetleg formális paraméterek, de ilyenek általánosan nincsenek jelen a bináris szinten. Az alacsonyszintű utasítások regisztereket, jelzőbiteket és memóriacímeket írnak és olvasnak, ennélfogva a létező megközelítéseket a megfelelő kifejezések használatával kell adaptálnunk.

Ahhoz, hogy az adat függőségeket feltérképezhessük, a program minden utasítását elemeznünk kell, és meg kell határoznunk, hogy mely regisztereket, jelzőbiteket olvassák, illetve írják. Az elemzés során természetesen nem vesszük figyelembe azon regisztert,

$$
\begin{aligned}
U_f^{(0)} &= \emptyset \\
U_f^{(i+1)} &= \bigcup_{j \in I_f} u_j \cup \bigcup_{g \in C_f} U_g^{(i)} \\
U_f &= U_f^{(i)}, \text{ ahol } U_f^{(i)} = U_f^{(i+1)}
\end{aligned}
$$

$$
\begin{aligned}
D_f^{(0)} &= \emptyset \\
D_f^{(i+1)} &= \bigcup_{j \in I_f} d_j \cup \bigcup_{g \in C_f} D_g^{(i)} \\
D_f &= D_f^{(i)}, \text{ ahol } D_f^{(i)} = D_f^{(i+1)}
\end{aligned}
$$

3.5. ábra. Az U_f és D_f halmazok meghatározása

amely a program vezérlését irányítja[5], mivel ennek szerepét már a CFG is tükrözi. A memória eléréseket is megpróbáljuk vizsgálni minden utasításnál, de a legtöbb esetben csak annyit tudunk egyelőre mondani, hogy az adott utasítás írja-e, illetve olvassa-e a memóriát. Azt, hogy pontosan melyik memóriacímet írja, vagy olvassa az utasítás, csak nagyon kevés esetben lehet pontosan meghatározni, ezért úgy kell kezelnünk minden ilyen utasítást, mintha az a teljes memóriát írhatná, illetve olvashatná. Ez persze nagyon sok nem létező adat függést hoz be. Ezen a pontatlanságon próbálunk javítani a 4.1. részben bemutatott eljárással, amelynek segítségével külön tudjuk választani azon utasításokat, melyek a verem memóriaterületét, illetve amelyek a vermen kívüli területeket használják.

Az utasítások elemzésekor a következő halmazokat adjuk meg: u_j halmaz tartalmazza a j utasítás által használt argumentumokat, d_j pedig a j által definiáltakat. Az elemzés során meghatározzuk minden j utasításra és minden $a \in d_j$-re azt az $u_j^{(a)}$ halmazt is, melynek elemei u_j azon elemei, amelyek a definiálásában szerepet játszanak. Nyilvánvalóan $u_j = \bigcup_{a \in d_j} u_j^{(a)}$ valamennyi j utasításnál. Elképzelhető, hogy olyan utasítással találkozunk, amelynek valamelyik a definiált argumentumára $u_j^{(a)} \subset u_j$. Magasszintű nyelveknél is elképzelhetőek ilyen utasítások, de ezek általában szétdarabolhatóak olyan részkifejezésekre, melyekben csak egy definiált argumentum van. Az alacsonyszintű nyelveknél ezt természetesen nem tudjuk megtenni. Ahhoz, hogy ezek az információk a PDG-ben is megjelenjenek, az utasításoknak megfelelő csomópontokból újabb csomópontokat származtatunk, amelyek a hozzájuk tartozó u és d halmazok elemeinek felelnek meg.

A magasszintű programoktól eltérően a bináris programok esetében az eljárások paraméter listája explicit módon nem definiált. Ahhoz, hogy ezt megadhassuk, egy interprocedurális elemzést kell végeznünk. Egy fixpont iterációt alkalmazunk annak érdekében, hogy meghatározhassuk valamennyi függvény bejövő és kimenő paramétere-

[5] Ezt a regisztert szokás program számlálónak *(PC - program counter)* nevezni.

$$C_{\texttt{add}} = \emptyset, \quad C_{\texttt{mul}} = \{\texttt{add}\}, \quad C_{\texttt{main}} = \{\texttt{add, mul, readin, writeout}\}$$

$$U_{\texttt{readin}} = U_{\texttt{writeout}} = D_{\texttt{readin}} = D_{\texttt{writeout}} = \{\texttt{R0 - R12, SP, LR, mem}\}$$

$U^{(0)}_{\texttt{add}}$	=	{R0, R1, LR}	$D^{(0)}_{\texttt{add}}$	=	{R0}
$U^{(0)}_{\texttt{mul}}$	=	{R0 - R5, SP, LR, mem}	$D^{(0)}_{\texttt{mul}}$	=	{R0 - R5, SP, LR, mem}
$U^{(0)}_{\texttt{main}}$	=	{R0, R4, R5, SP, LR, mem}	$D^{(0)}_{\texttt{main}}$	=	{R0, R1, R4, R5, SP, LR, mem}
$U^{(1)}_{\texttt{add}}$	=	$U^{(0)}_{\texttt{add}}$	$D^{(1)}_{\texttt{add}}$	=	$D^{(0)}_{\texttt{add}}$
$U^{(1)}_{\texttt{mul}}$	=	$U^{(0)}_{\texttt{mul}}$	$D^{(1)}_{\texttt{mul}}$	=	$D^{(0)}_{\texttt{mul}}$
$U^{(1)}_{\texttt{main}}$	=	{R0 - R5, SP, LR, mem}	$D^{(1)}_{\texttt{main}}$	=	{R0 - R5, SP, LR, mem}
$U^{(2)}_{\texttt{add}}$	=	$U^{(1)}_{\texttt{add}}$	$D^{(2)}_{\texttt{add}}$	=	$D^{(1)}_{\texttt{add}}$
$U^{(2)}_{\texttt{mul}}$	=	$U^{(1)}_{\texttt{mul}}$	$D^{(2)}_{\texttt{mul}}$	=	$D^{(1)}_{\texttt{mul}}$
$U^{(2)}_{\texttt{main}}$	=	$U^{(1)}_{\texttt{main}}$	$D^{(2)}_{\texttt{main}}$	=	$D^{(1)}_{\texttt{main}}$

3.6. ábra. A példa program U_f és D_f halmazainak meghatározása

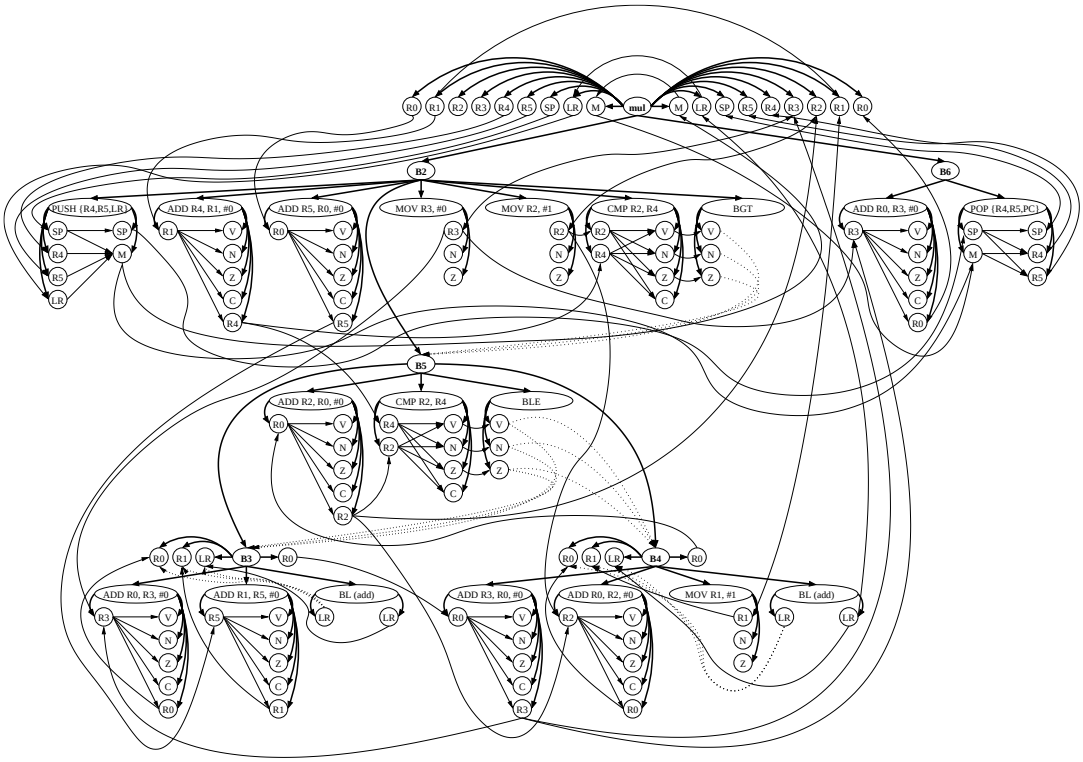
inek halmazát. Az f függvényre legyen U_f olyan halmaz, amelynek elemeit f használja, illetve D_f olyan, amelynek elemeit f definiálja[6]. Legyen továbbá I_f az f függvény utasításainak a halmaza, C_f pedig az f által hívott függvényeknek a halmaza. Az iteráció lépéseit az 3.5. ábra mutatja. Az iteráció utáni D_f halmaz elemei alkotják majd az f függvény kimenő paramétereit, míg $D_f \cup U_f$ elemei az f bejövő paramétereinek felelnek meg. Ezen paraméterek mint újabb csomópontok fognak megjelenni a gráfunkban, amelyek a függvény **entry** csomópontjához kapcsolódnak.

Az 3.6. ábra a példa program U_f és D_f halmazainak kiszámítási lépéseit foglalja össze. Az iterációk a `readin` és `writeout` függvényekre nincsenek részletezve, ezeknél csak a fixpontot adjuk meg.

A fenti elemzés után tehát a függvények **entry** csomópontjai további csomópontokkal egészülnek ki, amelyek a függvény által definiálható és használható argumentumoknak, vagyis a *formális bejövő* és *kimenő paraméterek*nek felelnek meg. Ezek után azon alapblokkokat is ki kell egészíteni újabb csomópontokkal, amelyekből függvényhívást kezdeményezünk. Értelemszerűen olyan csomópontokat kell felvenni, amelyek tükrözik azt, hogy a hívott függvények által milyen argumentumokat definiálhatunk, használhatunk. Az így felvett csomópontok felelnek majd meg az *aktuális bejövő* és *kimenő paraméterek*nek.

Miután felvettük a megfelelő formális bejövő és kimenő, valamint az aktuális bejövő és kimenő paramétereknek megfelelő csúcsokat, nekiállhatunk felvenni a megfelelő adat-

[6]Tulajdonképpen ezek a halmazok a HRB-algoritmus $GREF(f)$ és $GMOD(f)$ halmazainak felelnek meg [32].



3.7. ábra. A 3.3. ábrán ismertetett példa program `mul` függvényének PDG-je. A vastag élek a vezérlésfüggőségi, a vékony élek az adatfüggőségi, a szaggatott élek pedig a kompenzációs vezérlésfüggőségi élek, amelyek a hívó oldal és a hívott függvények közötti vezérlés függést reprezentálják.

függőségi éleket. Először azon adatfüggőségi éleket vesszük fel, amelyek egy utasításon belül jelentkeznek: az a argumentum definíciója a j utasításban adat függésben áll a j utasítás a' argumentumától, ha $a' \in u_j^{(a)}$. Ezután az utasítások közötti adat függések következnek: a j utasításban használt a argumentum adat függésben áll a k utasításbeli a argumentum definíciótól, ha létezik a CFG-ben k-ból j-be vezető út, amelynek során a-nak nincsen újabb definíciója. Ezen fenti megközelítés természetesen jól alkalmazható a regiszterekre és a jelzőbitekre, de a memória hivatkozásokra már nem, hisz az egész memóriát egy egyszerű argumentumként kezeljük. Az nem feltétlenül igaz, hogyha a memória valamely részét definiáljuk a k utasításban, akkor amikor majd a memóriát felhasználjuk a j utasításban, az pont azt a memóriacímet használja majd, amit a k. Ezért ha a j utasításban használjuk a memóriát, akkor meg kell keresnünk az összes elérhető memória definíciót minden lehetséges úton a CFG-ben, nem csak a legközelebbit. Ezzel persze nagyon sok olyan adat él kerül be a gráfba, ami a valóságban nem létezik. Mindenesetre a módszer biztonságos, és annak ellenére, hogy a sok felesleges adat él a programszelet optimális méretét növeli, nem fordulhat elő olyan eset, hogy két csomópont között adat függés van elvileg, de a gyakorlatban azt elveszítve a program szelet méretét hibásan csökkentenénk.

A PDG teljes felépítéséhez még mindig hiányzik néhány függőségi él. Egy függvény azon utasításainak használt argumentumaira, amelyek úgy érhetőek el a függvény **entry**-jéből indulva, hogy nem találunk hozzájuk definíciót, úgy kell tekintenünk, mintha a formális bejövő paraméterektől állnának adat függésben. Így nem veszítünk adat függést, hisz a formális bejövő paraméternek megfelelő aktuális bejövő paraméterhez keresett argumentum definíció az előbbi argumentumhoz közvetve tartozó definíció lesz. Ahhoz, hogy a szeletelés során meg tudjuk különböztetni a különböző hívási környezetekből származó függőségeket, az aktuális bemenő és kimenő paraméterek között meg kell határoznunk majd az összegző éleket, melyek meghatározását a 3.3.1 fejezetben tárgyaljuk. Végezetül tegyük fel, hogy egy alapblokk vezérlés függésben áll egy másik alapblokktól! Ez olyan, mintha az előbbi blokk az utóbbi utolsó utasításától függene, és így lényegében ezen utasítás argumentumaitól függ. Ezért a függő alapblokk valamennyi csomópontjába irányítunk egy vezérlésfüggőségi élt a másik blokk utolsó utasítása által használt argumentumoknak megfelelő csomópontokból. Ezeket az éleket *kompenzációs vezérlés függőségi él*eknek nevezzük. Hasonlóan a hívó blokkok aktuális bejövő paraméterei is függnek a blokk utolsó utasításának – ami nem más, mint a függvényt hívó utasítás – használt argumentumaitól.

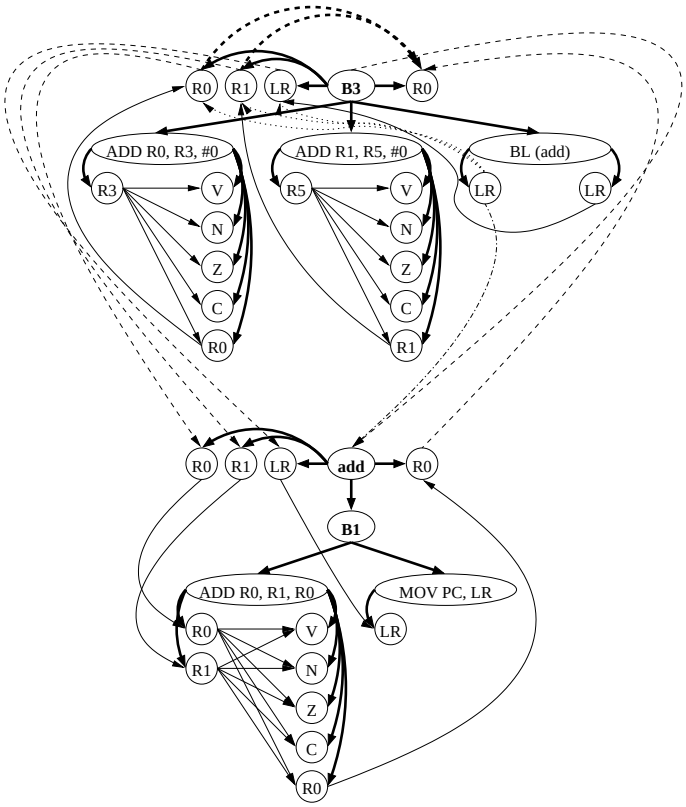
Az 3.7. ábra a példa program PDG-jének egy részletét mutatja. Az `Rx`, `LR` és `SP` csomópontok az egyes regisztereknek, a `C`, `N`, `V` és `Z` a jelzőbiteknek, míg `M` a memória hivatkozásnak felel meg.

3.3. Rendszerfüggőségi gráf meghatározása

Ahhoz, hogy a programot interprocedurálisan szeletelhessük, a program eddig gráf reprezentációját ki kell egészítenünk úgy, hogy a különálló függvények PDG-i egy egységes egészet alkossanak. Először is az aktuális bejövő és kimenő paramétereket össze kell kötnünk a nekik megfelelő formális bejövő és kimenő paraméterekkel az úgynevezett paraméter élek segítségével, majd az aktuális bejövő és kimenő paraméterek közötti függőségeket kell feltárnunk. Ennek eredményeként kapjuk meg az összegző éleket, amelyekkel kiegészítve a gráf reprezentációnkat, előáll a rendszerfüggőségi gráf, amely közvetlen alapot biztosít az interprocedurális szelet meghatározásához.

3.3.1. Összegző élek

Az interprocedurális szelet meghatározásához tehát az SDG-t kiegészítjük olyan élekkel, melyek az eljáráshívások által megvalósult függőségeket képesek kifejezni. Ezen összegző élek meghatározására Reps és társainak algoritmusát használtuk [52]. Ez az algoritmus a Horwitz, Reps és Binkley algoritmusának [32] továbbfejlesztett változata, amely algoritmus aszimptotikusan gyorsabb az eredetinél. Az alapötlete az algoritmusnak az, hogy



3.8. ábra. A 3.3. ábrához tartozó példa SDG-jének részlete a `add` függvény és a `mul` függvény `B3` blokkjának kiemelésével. A vékony és vastag folytonos élek az adat és vezérlés függéseket, a vastag szaggatott élek az összegző éleket, a vékony szaggatott élek a paraméter éleket, míg a pontozott élek a kompenzációs vezérlési és hívási éleket jelölik.

valamennyi P eljárásra meghatározza az azonos szinten megvalósuló utakat, amelyeket a P eljárás valamely formális kimenő csomópontja zár. Azon utak, amelyek P valamely formális bemenő csomópontjából indulnak, fogják bevezetni a megfelelő aktuális bemenő és kimenő paraméterek közötti összegző éleket valamennyi olyan hívási helynél, ahol a P eljárás hívható. A példaprogram SDG-jét a 3.8. ábra mutatja.

Az eddigi előkészületeinkkel nem okoz nehézséget az algoritmus implementálása bináris programok esetére. Oda kell azonban figyelni az **ismeretlen függvény**t hívó függvényhívásokra. Egy lehetséges megközelítés, ha a fel nem oldott függvényhívásokat úgy kezeljük, hogy a hívó oldal aktuális bejövő és kimenő paraméterei az **ismeretlen függvény**en keresztül elérhető összes függvény megfelelő formális bejövő és kimenő paramétereivel össze vannak kötve a paraméter élekkel.

Az SDG felépítésével már lehetőségünk van arra, hogy a vizsgált programunk bármely kritériumából programszeletet számítsunk Horwitz és társainak [32] algoritmusával, amely két menetben bejárva a rendszerfüggőségi gráfot határozza meg a programszelethez tartozó utasításokat.

A gyakorlat azt mutatja, hogy az így meghatározott szeletek még túlságosan nagyok. Ennek okainak vizsgálatával foglalkozunk a következő fejezetben, ahol megoldást is próbálunk adni a problémára.

3.4. Eredmények összegzése

Ebben a fejezetben megadtuk, hogy bináris programok szeleteléséhez használt függőségi gráfokat hogyan tudjuk előállítani. Összegyűjtöttük a függőségi gráfok felépítése közben jelentkező problémákat, amelyekre megoldási javaslatokat is adtunk. A vezérlésfüggőségi, adatfüggőségi és a rendszerfüggőségi gráfok felépítésének megadása a szerző önálló munkája.

4. fejezet

Bináris programok függőségi gráfjainak pontosítása

„Előbb csináld azt, ami szükséges, utána azt, ami lehetséges, és máris azt fogod csinálni, ami lehetetlen."

— Assisi Szent Ferenc

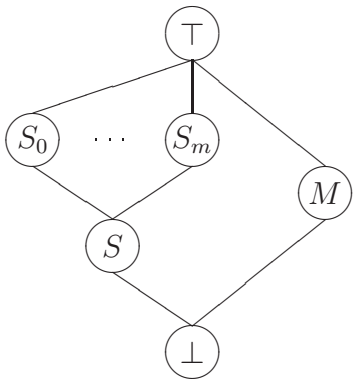
Habár a PDG és az SDG felépítésének eddig ismertetett módja biztonságos, azaz nem veszítünk el valódi vezérlés illetve adat éleket, mégis meglehetősen konzervatívnak mondható, főként az adat függőség konzervatív kezelése és az architektúrára jellemző információk hiánya miatt. Mind az adat függőségeket, mind a vezérlés függőségeket sokkal pontosabbá tehetjük, ha finomítjuk a statikus elemzést, vagy dinamikusan összegyűjtött információkat is felhasználunk a gráfok felépítésekor.

4.1. Pontosított PDG

Ez a fejezet olyan megközelítéseket mutat be, melyek az adatfüggőségi gráf pontosítását célozzák meg. Az egyik a függvény prologok és epilogok[1] egy heurisztikus vizsgálatán, míg a másik az utasítások memória használatának pontosításán alapszik.

A legtöbb jelenlegi architektúránál különböző függvényhívási konvenciók léteznek, amelyek meghatározzák, hogy egy adott függvény meghívásakor melyek azok a regiszterek, amelyeknek nem változhat meg a tartalma a függvényhívás által. A függvények ennek érdekében általában elmentik a regiszterek tartalmát a memóriába - rendszerint a veremre -, amikor a vezérlés a függvényre kerül, majd visszatérés előtt visszaállítják

[1]Míg a függvény prologban olyan információk kerülnek be, amelyek inicializálják és előkészítik az adott függvény végrehajtását, addig a függvény epilogok feladata a függvény futásának lezárása.



4.1. ábra. Háló a memóriahasználatok pontosításához

ezeket az értékeket. Ezeket a regiszter mentéseket és visszaállításokat könnyű meghatározni az architektúra és a hívási konvenció ismeretében.

Ha a mentett és visszaállított regiszterek halmazait meg tudtuk határozni, átdefiniálhatjuk a függvények kimenő paramétereinek a halmazát. Az f függvény kimenő paramétereinek a halmaza a továbbiakban a $D_f \backslash S_f$ halmaznak felel meg, ahol D_f ugyanaz, mint a 3.2.2. fejezetben, S_f pedig azon regiszterek halmaza, melyek a függvény hívása után elmentődnek és visszatérés előtt visszaállítódnak. Ezen új halmazokat felhasználva javítunk a függvényhívások konzervatívabb kezelésének pontosságán.

A PDG pontosításának másik lehetséges módja az adatfüggőségi analízis során használt memóriakezelés javítása. Bináris szinten a változók és függvény paraméterek magasszintű megvalósításai nem léteznek, a fordító regisztereket használ ezek helyettesítésére. A legtöbb architektúránál az elérhető regiszterek száma korlátozott, ráadásul a regiszterek tárolják a függvények egyes számításainak temporális eredményeit is. Így azon paramétereket és változókat, amelyeket egy adott pillanatban egyik regiszterhez sem tudjuk kötni, a memória egy meghatározott darabján tároljuk el. Ezen memóriahelyet szokás veremnek nevezni. Ha a vermen keresztül pontosabb adatfüggőségi elemzést tudnánk végezni, az jelentősen javítaná a szeletelés hatékonyságát is.

Alkalmazásunkban valamennyi utasításnál meghatározzuk, hogy az egyes regiszterek milyenek lehetnek a tartalmukra vonatkozó statikusan elérhető információk alapján az utasítás végrehajtása előtt, illetve annak végrehajtása után. Ehhez a jellemzéshez egy hálót vezettünk be, amelyet az 4.1. ábra mutat. Minden utasításnál az egyes regisztereket a háló elemeiből álló párok fogják jellemezni az utasítás végrehajtása előtti és utána levő tartalmuknak megfelelően.

Ha a $\top$ elemmel jelölünk egy regisztert, az azt jelenti, hogy a regiszter még tartalmazhat a későbbiekben memóriahivatkozást. A $\bot$ jelentése az, hogy statikusan nem lehetett eldönteni, hogy a regiszter veremhivatkozást tartalmaz-e, vagy sem. Ilyen esetben lehet, hogy a veremre, de az is lehet, hogy a vermen kívüli memóriaterületre hi-

$$
\begin{aligned}
\text{bármi} \sqcap \top &= \text{bármi} \\
\text{bármi} \sqcap \bot &= \bot \\
S_i \sqcap S_j &= S_i, \text{ ha } i = j \\
S_i \sqcap S_j &= S, \text{ ha } i \neq j \\
S_i \sqcap S &= S \\
S_i \sqcap M &= \bot \\
S \sqcap M &= \bot
\end{aligned}
$$

4.2. ábra. A memóriahasználatot jellemző háló metszet művelete

vatkozik. Az M jelölés azt érzékelteti, hogy a regiszter olyan memóriahelyre hivatkozik, ami nem eleme a veremnek. Amíg egy S-sel jelölt elem olyan hivatkozást tartalmaz, amely a verem valamely pozíciójára hivatkozik, de statikusan nem tudjuk pontosan, hogy melyikre, addig az S_i jelölés egy konkrét verem helyet határoz meg.

Az algoritmus ezen statikusan elérhető információkat a CFG-t felhasználva egy iteratív eljárással adja meg a következő módon[2]:

1. A függvény minden utasításánál a regiszterekhez a $(\top, \top)$ jelölést párosítsuk! Az első utasítás regisztereihez a $(\bot, \top)$ párt kapcsoljuk! Ez alól kivétel a verem mutatónak megfelelő regiszter, amelyhez az $(S_0, \top)$ pár tartozik!

2. A függvény első utasítását tegyük a W halmazba!

3. Vegyünk ki W-ből egy utasítást! Aktualizáljuk az utasítás végrehajtása előtti regiszterértékekhez kapcsolt háló elemeket a megelőző utasítás végrehajtása utáni megfelelő regiszterértékekhez kapcsolt hálóelemek és a 4.2. ábrán megadott $\sqcap$ szabály alapján. Az új bejövő értékek függvényében értékeljük ki az utasítást, és határozzuk meg, hogy milyen lehet az egyes regiszterek tartalma. Amennyiben az utasítás végrehajtása után bármely regiszterhez számolt érték módosítja az eredetileg neki megfelelő hálóelem értékét, úgy az utasítást közvetlenül követő utasításokat tegyük be a W halmazba!

4. Ha W nem üres, akkor lépjünk a 3. pontra!

Ezen eljárás eredményét felhasználhatjuk az adat élek meghatározásánál. Módosítva a 3.2.2. részben bemutatott adatfüggőségi elemzést, csökkenteni tudjuk azon adat élek számát, melyek mögött valódi adat függés nincs. A konzervatív megközelítésben a teljes memóriát egyetlen argumentumként ábrázoljuk, de felhasználva a fenti elemzést, differenciálni tudjuk az egyes memórahivatkozásokat, és a háló elemeinek megfelelően címkézve

[2]Mivel a háló magassága véges, ráadásul az algoritmus által alkalmazott $\sqcap$ művelet monoton, így az algoritmus biztosan terminál [49].

```
            add:
00002ECC    ADD R0, R1, R0          B1
00002ECE    MOV PC, LR
            mul:
00002ED0    PUSH {R4,R5,LR}         B2
00002ED2    ADD R4, R1, #0
00002ED4    ADD R5, R0, #0
00002ED6    MOV R3, #0
00002ED8    MOV R2, #1
00002EDA    CMP R2, R4
00002EDC    BGT 0x00002EF6
00002EDE    ADD R0, R3, #0          B3
00002EE0    ADD R1, R5, #0
00002EE2    BL 0x00002ECC (add)
00002EE6    ADD R3, R0, #0          B4
00002EE8    ADD R0, R2, #0
00002EEA    MOV R1, #1
00002EEC    BL 0x00002ECC (add)
00002EF0    ADD R2, R0, #0          B5
00002EF2    CMP R2, R4
00002EF4    BLE 0x00002EDE
00002EF6    ADD R0, R3, #0          B6
00002EF8    POP {R4,R5,PC}
            main:
00002EFA    PUSH {R4,R5,LR}         B7
00002EFC    ADD SP, #-8
00002EFE    BL 0x00002EAC (readin)
00002F02    ADD R5, R0, #0          B8
00002F04    MOV R0, #0
00002F06    STR R0, [SP, #0]
00002F08    MOV R0, #1
00002F0A    STR R0, [SP, #4]
00002F0C    MOV R4, #1
00002F0E    CMP R4, R5
00002F10    BGE 0x00002F2A
00002F12    LDR R0, [SP, #0]        B9
00002F14    ADD R1, R4, #0
00002F16    BL 0x00002ECC (add)
00002F1A    STR R0, [SP, #0]        B10
00002F1C    LDR R0, [SP, #4]
00002F1E    BL 0x00002ED0 (mul)
00002F22    STR R0, [SP, #4]        B11
00002F24    ADD R4, #1
00002F26    CMP R4, R5
00002F28    BLT 0x00002F12
00002F2A    LDR R0, [SP, #0]        B12
00002F2C    LDR R1, [SP, #4]
00002F2E    BL 0x00002EBE (writeout)
00002F32    ADD SP, #8              B13
00002F34    POP {R4,R5,PC}
```

4.3. ábra. A 4.3. ábra programjának interprocedurális szelete a 00002F2A memóriacíme található `LDR R0, [SP, #0]` utasítás `R0` regiszteréből kiindulva. A fekete betűkkel szedett utasítások bele tartoznak a szeletbe, míg a szürkével szedettek nem.

különböző argumentumként jelennek meg a különböző típusú memória elérések. Ezek után már csak azt kell definiálni, hogy az egyes memória hivatkozó csomópontok mikor állhatnak egymással adat függésben. A j utasítás által használt a memória hivatkozás adat függésben áll a k utasítás által definiált a' memória definiálástól, ha $a' \sqcap a \in \{a, a'\}$. Amennyiben a' valamelyik S_i-vel egyezik meg, akkor létezik a CFG-ben k-ból j-be vezető út úgy, hogy azon a'-t nem definiáljuk újra.

Megközelítésünkben a memória tökéletesített kezelését nem terjesztettük ki az aktuális - és azokon keresztül a formális - bejövő és kimenő paraméterekre a verem- és a memória elérések interprocedurális elemzésének nehézségei miatt. Így azzal a pesszimista megközelítéssel éltünk, hogy az aktuális – és így a formális – bejövő és kimenő paramétereknél a $\perp$ típusú memória hivatkozás jelenik meg.

Felhasználva a fejezetben mutatott javításokat, a 4.3. ábra az eredeti kód azon utasításait mutatja, amelyek a 00002F2A utasítás `R0` regiszteréből indított hátrahaladó statikus szeletbe tartoznak. Amennyiben nem alkalmaznánk a bemutatott pontosításokat, úgy a teljes `mul` függvény és a B8, B10 és B11 alapblokkok valamennyi utasítása a szelet részét képeznék.

4.2. Hívási gráf pontosítása dinamikus információk alapján

Érdemes elgondolkodni azon, hogy az eddig megvalósított javítások mellett mit lehetne még figyelembe venni ahhoz, hogy a szelet mérete tovább csökkenhessen. Mock és társai points-to információkat használtak C programok statikus szeleteinek csökkentésére [48]. Azzal, hogy dinamikusan gyűjtött adatokkal pontosították a hívási gráf méretét, és a szeleteléshez ezt a leredukált gráfot használták fel, sokkal kisebb szelet méreteket kaptak. Az eredményeik rávilágítottak arra, hogy különösen azoknál a programoknál csökkentek a szelet méretek, amely programok nagy számban használtak indirekt függvényhívásokat. Természetesen ez a módszer már nem tekinthető statikusnak, illetve biztonságosnak, hiszen lehet olyan eset, amikor egy függőség nem jelenik meg a szeletben. Ennek ellenére hasznos lehet olyan esetekben, amikor nem célunk egy tökéletes szelet meghatározása. Ráadásul kétségtelen előnye a módszernek az egyszerűsége, és könnyű kivitelezhetősége.

Bináris programok esetében különösen a nagyobb programoknál számos esetben találunk statikusan fel nem oldott függvényhívásokat. Ezek jelenléte miatt így itt is számos olyan függés kerül be a gráfba, amely gyakorlatilag nem valósul meg. Ezeknek a hatásait a szelet méretére vonatkozóan úgy tudjuk tesztelni, ha dinamikusan összegyűjtött adatok alapján pontosítjuk a hívási, illetve a vezérlési gráfunkat.

Miután a CFG-t felépítettük, gyűjtsük ki az összes, statikusan fel nem oldott indirekt függvényhívás címét! Ezután reprezentatív bemenetekre futtassuk le a programjainkat egy vezérelhető eszköz segítségével! Az előbb meghatározott címeknél, mint töréspontoknál a regiszterek aktuális értékeit gyűjtsük ki egy log-fájlba! Ilyen vezérelhető környezet lehet akár egy szoftver emulátor, vagy egy valós gép egy megfelelő nyomkövető felülettel.

A generált log-fájl segítségével meghatározhatjuk a statikusan fel nem oldott indirekt függvényhívások pillanatnyi céljait, és így az **ismeretlen függvény** csomópontba irányuló hívási éleket felcserélhetjük az aktuális célokhoz vezető hívási élekkel. Azokat a hívási helyeket, amelyekre az adott program egyik meghívása során sem került a vezérlés, különbözőképpen kezelhetjük. Egyik megközelítés szerint úgy vesszük, mintha ezeknél a pontoknál nem történne semmilyen függvényhívás. Ez a megközelítés azonban túl optimista szeletet eredményez. A másik lehetőség, hogy az ilyen helyeknél meghagyjuk az **ismeretlen függvény** csomópont meghívását. Az előbbi esetet nevezzük agresszív dinamikus megközelítésnek, mert itt tudjuk legnagyobb mértékben csökkenteni a függőségeink számát, utóbbi esetet pedig nevezzük kombinált dinamikus közelítésnek, hisz itt a tiszta statikus módszer konzervatív megközelítése keveredik a dinamikus módszerrel.

A dinamikus információk összegyűjtése után az összegző éleket újra meg kell adnunk az SDG-ben, illetve az interprocedurális szeleteket a szokásos módon megint meg kell határozni. Ne felejtsük el, hogy a dinamikusan összegyűjtött információk használatával

a meghatározott szeletek már nem tekinthetőek statikusnak, és az is elképzelhető, hogy nem lesz biztonságos ez a szeletelés, de elegendő számú tesztesettel jó közelítést adhatunk a statikus vezérlési gráfra és azon keresztül a statikus interprocedurális szeletek méretére!

4.3. Eredmények összegzése

A fejezetben publikált eredmények célja, hogy a felépített függőségi gráfok a lehető legjobban tükrözzék a rendszerben valóban jelen levő függőségeket. A függőségek statikus módszerekkel való pontosításai révén a megadott függőségek konzervatívak maradnak, azaz csak olyan függőségi élek elhagyása megengedett, amelyek mögött valódi függés nincs. A dinamikus pontosításokkal a függőségi élek bár már nem feltétlenül tartalmazzák a programban létező összes függést, használatuk célravezető és elegendő lehet bizonyos alkalmazásokban. A szerző önálló eredménye az adatfüggőségi háló révén megvalósított pontosított adatfüggőségi elemzés, míg a többi pontosításának a megtervezése a szerzőtársakkal oszthatatlan eredménynek minősül.

5. fejezet

Gyakorlati eredmények

„Elvárjuk a kutatótól, hogy bizonyítékokkal szolgáljon. Ha például egy nagy hegy felfedezéséről van szó, azt várjuk el tőle, hogy nagy köveket hozzon magával."

— *Antoine de Saint-Exupéry*

Ebben a fejezetben kiértékeljük az egyes módszereinket, amiket ebben a részben bemutattunk. Az eddig ismertetett eljárásokat statikusan linkelt ARM programok elemzéséhez implementáltuk, majd a SPEC CINT2000 [56] és Media Bench benchmark suit [45] programjain értékeltük ki. A kiválasztott programokat a Thumb utasításkészletű ARM7T processzorra fordítottuk a Texas Instruments TMS470R1x Optimalizált C fordítójának 1.27e verziójával.

5.1. Gyakorlati mérések ismertetése

Mielőtt a bemutatott módszereink tényleges összehasonlításába fogunk, ismertetjük a méréseink alapjául szolgáló binárisok néhány jellemzőjét. Az 5.1. táblázatban megadjuk, hogy az egyes programok mekkorák, mennyi forráskódból származó utasítást és mennyi könyvtári függvényből származó utasítást tartalmaz a bináris kódjuk, valamint az eredeti C kódjaik hány sorosak. A bináris kódban megtalálható, valamint a ténylegesen elérhető függvények számát, illetve a dinamikus esetekkel elért függvények számát ismerteti az 5.2. táblázat, míg az 5.3. táblázat az indirekt hívási helyekről és az itt hívható függvények számáról mutat képet. A táblázatokban az oszlopok első elemei a forrás azon részét jellemzik, amik az elemzett rendszer részei, míg a zárójelbe tett adatok a programhoz csak hozzálinkelt könyvtári függvények jellemzőit összegzik. Az, hogy a statikusan elérhető függvények száma nem egyezik meg teljesen a program függvényeinek a számával, a linkelés pontatlanságának tudható be. Az elért, illetve futtatott függvények száma azt

5.1. táblázat. Program méretek

Program	Bináris méret (byte)	Utasítások száma	Forrás sorok száma
ansi2knr	12 596	774 (5 014)	693
decode	15 476	2074 (5 162)	1 593
bzip2	43 324	8 788 (5 311)	4 247
toast	37 748	10 662 (5 517)	5 997
sed	42 332	13 284 (5 820)	12 241
cjpeg	99 352	37 019 (7 482)	28 720
osdemo	419 032	177 214 (7 025)	62 374

5.2. táblázat. Függvények száma

Program	Összes függvény	Elérhető függvények	Tesztek által elért függvények
ansi2knr	5 (114)	5 (97)	4 (57)
decode	26 (109)	21 (91)	21 (45)
bzip2	73 (119)	51 (102)	38 (53)
toast	86 (124)	72 (107)	60 (54)
sed	102 (142)	92 (128)	57 (70)
cjpeg	381 (149)	327 (133)	134 (62)
osdemo	1 186 (145)	675 (127)	122 (85)

mutatja, hogy néhány esetben az alap tesztesetek elég jó kódlefedettséget biztosítanak, de néhol, különösen a nagyobb programoknál az elért lefedettség érték meglehetősen alacsony. Néhol, mint például az `osdemo` programnál, nem is kaphatnánk jobb lefedettséget, még akkor sem, ha a statikus analízis szerint ez lehetséges lenne. Tipikus oka ennek a szituációnak az indirekt hívások számának magas értéke.

Gyakorlati méréseinkben öt esetet vizsgálunk az eddigi fejezeteknek megfelelően. A megoldások egymásra épülnek, azaz a méréseink valamennyi pontosítást magukban foglalják, amik az adott módszert megelőzik a következő listában. A továbbiakban az egyes módszerekre vagy a listában megadott sorszámukkal, vagy vastagon szedett rövid nevükkel hivatkozunk.

1.) **Konzervatív** módszernek nevezzük a továbbiakban az eredeti eljárásunkat, ahol az 3.1., 3.2. és 3.3.1. fejezetek alapján felépített függőségi gráfokon végezzük a szeletelést.

2.) **Regiszter mentés**es módszer az, ahol a függvényszintű információk segítségével csökkentjük az egyes függvények által módosított regiszterek számát, amivel csökkentjük az összegző élek számát, és ezen keresztül az adat élek számát.

3.) A **verem elemzés** során a 4.1. fejezetben bevezetett háló segítségével megkülönböztetjük a memória eléréseket, ezzel pedig redukáljuk az adat éleket.

5.3. táblázat. Indirekt függvény hívások

Program	Hívási helyek	Hívható függvények
ansi2knr	0 (12)	0 (12)
decode	1 (12)	3 (10)
bzip2	0 (12)	0 (10)
toast	6 (12)	13 (12)
sed	1 (12)	3 (16)
cjpeg	469 (32)	181 (15)
osdemo	245 (12)	359 (14)

5.4. táblázat. Adatfüggőségi élek számának alakulása

Program	1.	2.	3.	4.	5.
ansi2knr	71 394	62 533	49 807	48 965	47 866
decode	79 514	70 472	56 352	54 972	53 072
bzip2	231 338	199 596	161 773	159 822	157 271
toast	203 928	162 016	134 694	131 846	127 117
sed	867 064	562 496	486 080	482 758	480 999
cjpeg	768 188	655 530	513 570	506 729	402 248
osdemo	5 394 204	4 832 597	3 398 766	3 391 853	3 322 974

4.) A **kombinált dinamikus** módszer esetében a hívási gráfot az 4.2. fejezet szerint gyűjtött dinamikus információk alapján azon függvényhívási pontoknál pontosítjuk, amelyek valamely teszteset által érintettek voltak.

5.) Az **agresszív dinamikus** módszer esetében valamennyi indirekt függvényhívást elhagyjuk, és a hívási gráfot csak ott egészítjük ki a megfelelő hívási éllel, ahol annak célpontját a dinamikus adatok segítségével egyértelműen be tudjuk azonosítani.

A pontosítási módszerek mindegyike a programszeletelést azzal teszik hatékonyabbá, hogy valamilyen módon redukálják a rendszerfüggőségi gráfot, amely megfelelő bejárása így kisebb szeletet eredményez. A legjelentősebb változások az adat, összegző és hívási élek számának csökkenésében jelentkeznek, így ezeket vizsgáljuk meg alaposabban. Valamennyi módszernél változik az adat függőségek száma. Míg a statikus javítások közvetlen hatnak erre a változásra, amely egyébként kihat az összegző élek változására is, addig a dinamikus javítások a hívási gráf redukálásával, és az összegző élek számának csökkenésével hatnak az adat függőségek számának csökkenésére. Az 5.4. táblázat az adat függések számát, az 5.5. táblázat az összegző élek számát, míg az 5.6. táblázat a hívási élek számát összegzi az egyes módszerek esetében.

Ahhoz, hogy összehasonlíthassuk a különböző módszereinket, valamennyi módon elő kell állítanunk a programokhoz tartozó rendszerfüggőségi gráfokat. Az összehasonlíthatóság végett ugyanazon kritériumokból indítva számítunk szeleteket a különböző esetekben. Szeletelési kritériumként azon függvényeknek az utasításait használtuk fel, amelyekre valamely dinamikus futtatás során is rákerült a vezérlés.

5.5. táblázat. Összegző élek száma

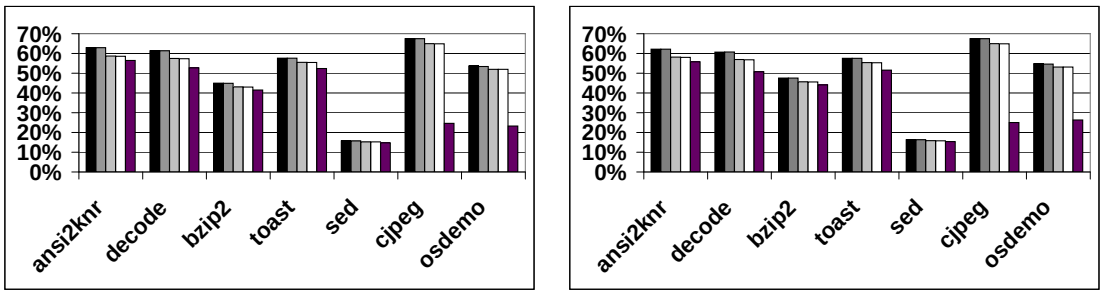
Program	1.	2.	3.	4.	5.
ansi2knr	19 859	19 321	9 626	8 852	8 032
decode	22 864	22 356	11 099	9 730	8 312
bzip2	61 404	60 280	30 363	28 536	26 608
toast	48 061	47 402	24 500	21 740	18 104
sed	69 924	69 281	37 082	34 174	32 781
cjpeg	228 210	220 657	118 127	111 278	42 717
osdemo	779 437	682 889	327 126	320 846	278 559

5.6. táblázat. Hívási élek

Program	1., 2., 3.	4.	5.
ansi2knr	401 (144)	324 (67)	264 (7)
decode	428 (169)	358 (99)	267 (8)
bzip2	781 (120)	736 (75)	666 (5)
toast	1 010 (450)	774 (214)	574 (14)
sed	1 047 (247)	886 (86)	810 (10)
cjpeg	99 320 (98 196)	83 145 (82 021)	1 217 (93)
osdemo	106 819 (95 861)	91 940 (80 982)	10 999 (41)

Azért, hogy a kritériumok kiválasztásából adódó torzításokat elkerüljük, ezen függvények valamennyi utasítására meghatározzuk a programszeleteket. Kétféle mérést teszünk, az egyikben kritériumként csak olyan utasítást választunk, amelyek az eredeti forrásból származnak, míg a másiknál kritériumként felvesszük a könyvtári függvények utasításait is. Az előbbi eset átlagos szelet méreteit az 5.7. táblázat, míg a második eset átlagos szelet méreteit az 5.8. táblázat foglalja össze, illetve az 5.1. ábra teszi őket áttekinthetőbbé. Az 5.9. táblázat második oszlopa megadja, hogy az első esetben, azaz a 5.7. táblázatban feltüntetett méréseknél, hány kritériumot használtunk, míg a harmadik oszlop azt, hogy a 5.8. táblázat által összegzett mérésekben mennyi kritériumunk volt.

Magasszintű nyelveknél jellemzően az a célunk, hogy a saját kódunk utasításaiból indítsunk szeleteket. Bináris esetben azonban elképzelhető olyan szituáció - például egy



5.1. ábra. Az 5.7. és 5.8. táblázatok adatainak grafikus összehasonlítása

5.7. táblázat. Átlagos utasításszám a programszeletekben, amely szeletek kritériumai az eredeti forráshoz tartozó utasítások voltak.

Program	1.	2.	3.	4.	5.
ansi2knr	495 (3 146)	495 (3 146)	485 (2 911)	485 (2 904)	485 (2 782)
decode	1 205 (3 234)	1 205 (3 234)	1 167 (2 992)	1 166 (2 984)	1 097 (2 719)
bzip2	3 131 (3 205)	3 131 (3 205)	3 084 (2 985)	3 084 (2 977)	3 084 (2 764)
toast	5 819 (3 507)	5 819 (3 507)	5 695 (3 287)	5 693 (3 277)	5 373 (3 107)
sed	7 378 (3 859)	7 378 (3 859)	7 282 (3 619)	7 281 (3 604)	7 159 (3 397)
cjpeg	24 863 (5 180)	24 863 (5 180)	24 038 (4 873)	24 003 (4 865)	6 700 (4 246)
osdemo	94 709 (4 167)	94 238 (4 145)	91 855 (3 890)	91 825 (3 876)	39 398 (3 440)

5.8. táblázat. Átlagos utasításszám a programszeletekben, amely szeletek kritériumai az eredeti forrás és a könyvtári függvények utasításai.

Program	1.	2.	3.	4.	5.
ansi2knr	478 (3 121)	478 (3 121)	469 (2 897)	469 (2 889)	462 (2 765)
decode	1 191 (3 202)	1 191 (3 202)	1 153 (2 966)	1 152 (2 959)	972 (2 701)
bzip2	3 493 (3 211)	3 493 (3 211)	3 441 (2 990)	3 441 (2 982)	3 435 (2 776)
toast	5 807 (3 505)	5 807 (3 505)	5 677 (3 282)	5 675 (3 272)	5 253 (3 091)
sed	7 776 (3 873)	7 776 (3 873)	7 675 (3 632)	7 674 (3 617)	7 565 (3 416)
cjpeg	3 493 (3 211)	3 493 (3 211)	3 441 (2 990)	3 441 (2 982)	3 435 (2 776)
osdemo	96 762 (4 252)	96 372 (4 235)	93 916 (3 974)	93 889 (3 961)	44 974 (3 540)

linkelés utáni módosítás -, amikor a könyvtári függvények utasításainak vizsgálata éppen olyan fontos, mint az eredeti kódhoz tartozó utasításoké. Mivel mind a két esetben a szelet méretek változásai hasonló trendet mutatnak, így függetlenül az utasítás helyétől, általános észrevételeket vonhatunk le az egyes javítási módszerekről. A szelet méreteket részletesebben vizsgálva elmondhatjuk, hogy minden pontosítás nélkül is az átlagos szelet méret az eredeti program 52%-a, ami azt igazolja, hogy a statikus szeletelés hatékony eszköz lehet bináris programokat elemző számos alkalmazásban.

5.2. Statikus javítások hatásainak összegzése

A statikus módszereink közül a regiszter mentéses módszer, amelyik csak a mentett regiszterek figyelésével javítja az adatfüggőségi gráfunkat, olyan kis mértékben csökkentette a szelet méreteket, hogy az átlagos szelet méretek szinte nem változtak. Ahogyan az a szelet méreteket bemutató ábrákról leolvasható, az átlagos szelet méretek a konzervatív szelet méretekhez képest el sem térnek csak egy esetben, az `osdemo` programnál. Nem meglepő ez az eredmény a kísérletünkben, ha figyelembe vesszük azt a tényt, hogy a tesztelt bináris programok utasításai 16 bites Thumb utasításokból állnak. A Thumb a 32 bites ARM architektúra kiterjesztése. A Thumb utasítások megvalósítják a leggyakrabban használt 32 bites ARM utasításokat, amelyeket így 16 bit széles opcode-

5.9. táblázat. Kritériumok száma az 5.7. és 5.8. táblázat méréseiben

Program	Forráshoz tartozó kritériumok száma	Forráshoz és könytári függvényekhez tartozó kritériumok száma
ansi2knr	761	2 894
decode	1 670	3 445
bzip2	8 237	10 112
toast	7 428	9 631
sed	11 218	14 431
cjpeg	12 103	14 640
osdemo	20 142	24 319

okkal tömörítenek. Így egy futtatás során ezek a 16 bites utasítások lehetővé teszik az eredeti 32 bites utasítás transzparens végrehajtását annélkül, hogy ez teljesítménybeli romlást okozna. Thumb utasításoknál a regisztert specifikáló mezők tipikusan 3 bit hosszúságúak, míg az ARM utasításoknál ugyanez 4 bit. Ennek köszönhetően a Thumb utasításokban az `R0`, ..., `R7` regiszterek használata a jellemző, és a maradék 8 regiszter csak néhány utasításnál (például a `MOV` utasításnál) érhetőek el. Ráadásul az összesen 16 regiszterből három darab speciális funkciókkal bír. Ezek együttesen azt eredményezik, hogy nagyon sok esetben kell a regiszterek pillanatnyi tartalmát eltárolni a memóriában, illetve átmozgatni más regiszterekbe. Tovább csökkenti a módszerünk hatékonyságát, hogy a Thumb kódoknál az első 4 darab regiszter soha nem kerül mentésre. Indirekt függvényhívásoknál pedig egy regiszterről csak akkor tehetjük fel, hogy mentett regiszter, ha az indirekt hívás helyén hívható valamennyi függvény menti azt.

Habár az adat függőségek száma jelentősen csökkent ebben az esetben (10,4% és 35,1% között), az összegző élek száma minimálisan változott. A kisebb programoknál csupán 1,3% - 3,3% közötti értékkel, míg az `osdemo` programnál is csak 12,4%-kal. Az, hogy az `osdemo` programnál van érzékelhető különbség a konzervatív mérés és a regiszter mentéses módszer szeletei között, az összegző élek kicsit nagyobb változásának köszönhető.

A verem elemzéses technikát alkalmazva az adatfüggőségi élek számát csökkentjük elsődlegesen (13,6% és 29,7% közötti értékkel), amely változások maguk után vonják az összegző élek igen jelentős redukcióját (46,5% és 52,1%-os csökkenését). Ezek a változásoknak már mérhető (1,5% és 4% közötti) szelet méret csökkenést jelentenek, bár azért ez sem tekinthető olyan nagy javulásnak. Az, hogy ezek a változások sem olyan nagyok, tulajdonképpen annak köszönhető, hogy interprocedurális szintre nem terjesztettük ki a regiszter tartalmának figyelését, így valamennyi eljárásban a formális bejövő paraméterekről azt tesszük fel, hogy azok tartalma a memória tetszőleges pontjáról származhat, és ugyanígy kezeljük valamennyi hívási hely aktuális kimenő paraméterének tartalmát is.

5.3. Dinamikus javítások hatásainak összegzése

Ahhoz, hogy az indirekt függvényhívásokra vonatkozó dinamikus adatokat összegyűjtsük, a kiválasztott programokat a Texas instruments TMS470R1x C Source Debugger emulátorával futtattuk. Ahhoz, hogy a lehető legjobb kódlefedettséget érhessünk el, a programok eredeti teszt bemeneteit használtuk fel. Az 5.2. táblázatban megadtuk a bináris kódban fellelhető függvények számát, illetve azt, hogy ezek közül mennyi a statikusan elérhető függvények száma, illetve hogy a tesztekkel valójában hány függvényt érintünk. Több esetben egész jó kód lefedettséget értünk el, de vannak olyan esetek is, különösen a nagyobb programoknál, ahol lehetetlen nagyobb lefedettséget elérni, hiszen a kód nagy részéről belátható, hogy az valójában sohasem fog lefutni, még ha a statikus analízis alapján ez lehetséges is lenne. Tipikusan ez azokban az esetekben fordul elő, ahol az indirekten hívható függvények száma nagy.

A dinamikus megoldásokkal leginkább a hívási élek számát csökkenthetjük, ahogy azt az 5.6. táblázatban már megfigyelhettük. A kombinált dinamikus módszerben a hívási élek száma 10%-32%-kal, míg az agresszív dinamikus módszer esetén 25%-99%-kal csökken a konzervatív, illetve a statikus esetekhez képest. Az agresszív módszernél a `cjpeg` és `osdemo` programok hívási éleinek drasztikus csökkenése egyrészt a nagy számú indirekt hívásoknak, másrészt pedig annak tudható be, hogy ezen programok tesztlefedettsége volt a legrosszabb. Ha azt vesszük, hogy a kombinált dinamikus módszer esetében a hívási élek redukciója ezeknél a programnál csak 16,4% és 14,7%, akkor megállapíthatjuk, hogy a legtöbb indirekt hívási helyet el sem értük a tesztek futtatásával. Ennek ellenére nem meglepő, hogy a dinamikus adatokkal való pontosításoknál azokban az esetekben, ahol az indirekt függvényhívások száma relatív kevés, ott sokkal kisebb mértékben csökken a hívási élek száma. Elsősorban ennek köszönhető az is, hogy a legtöbb esetben a dinamikus pontosításnál az összegző élek száma sem csökken olyan látványosan, valamint annak, hogy bár a hívási gráfot leredukáltuk, ezzel az esetek többségében csupán olyan könyvtári függvények meghívását kerültük el, amelyek hasonlóan viselkednek - legalábbis regiszter használat terén - azzal a függvénnyel, amely meghívását végül betettük a gráfba.

Habár a statikus esetekhez képest az adatfüggőségek száma és az összegző élek száma is csökken a dinamikus esetekben, ezek mértéke nem olyan nagy, mint a verem elemzéses technika esetében. A dinamikus esetekben így egyértelmű, hogy a szelet méretek csökkenése a hívási élek redukciójának köszönhető inkább.

Általánosítva a kapott eredményeket elmondhatjuk, hogy bármely redukciója a függőségi gráfoknak a szelet méretek csökkenését okozzák. Nem meglepően azok a függőségek, amelyek magasabb szinten jellemzik a rendszerünket – azaz jelen esetben a hívási kapcsolatok – lesznek azok, amelyeknek a legnagyobb hatása van a kapott szelet méretre.

5.4. Eredmények összegzése

A fejezet összefoglalja az ismertetett módszerek gyakorlati megvalósításainak eredményeit. A konzervatív szeletelési technikával elért 52%-os átlagos szelet méret azt igazolja, hogy a statikus szeletelés bináris programok esetében is hatékonyan segítheti számos rá épülő alkalmazást. A különböző pontosításokkal pedig még további szelet méret csökkenést érhetünk el. A mérések megtervezése, és a mérési eredmények kiértékelése a társszerzőkkel közös, oszthatatlan eredményeknek minősülnek. A vezérlési folyam gráf előállításának implemetálásán kívül valamennyi függőségi gráf gyakorlatban való megvalósítása, illetve az ismertetett módszerek implementálása a szerző munkája.

II. rész

Statikus Execute After és Statikus Execute Before relációk

6. fejezet

Bevezető

„Minden mindenből ered, és minden mindenné válik."

— Leonardo Da Vinci

A program komponensei közötti függőségek felismerésének fontosságát igen gyakran szemléltetik olyan drasztikus példákkal, ahol egy szoftvernek köszönhető katasztrófát a program komponensei között lévő fel nem fedezett függőségek okoztak. Ilyenre példa az Ariane-5 hordozórakéta első tesztrepülése (Ariane-5 Flight 501), ahol az indítás után 37 másodperccel a rakéta önmegsemmisítést hajtott végre az irányító szoftver hibája miatt. Talán ez volt az eddigi történelem egyik legdrágább számítógépes „meghibásodása". A hibát egy olyan lekezeletlen kivétel okozta, amely akkor lépett fel, amikor egy 64 bites adat 16 bitre történő konvertálása nem sikerült annak túl nagy mérete miatt [35]. Amennyiben még a program tesztelése során felfedezték volna, hogy az önmegsemmisítés végrehajtása függ attól, hogy a hiba nem lett kezelve, a tragédia elkerülhető lett volna.

A szoftverfejlesztésnek számos olyan területe létezik, amelyek a programban fellelhető függőségek elemzésén alapszanak. Ilyenek többek között a programmegértés, a karbantartás, a változás propagáció, a hatásanalízis, a tesztelés. A szoftverfejlesztés során, ahogy egy program fejlődik, és egyre összetettebbé válik, úgy lesz a program egyre kevésbé átlátható, érthető, és egyre nehezebb lesz meghatározni azt is, hogy bármely változtatás a programban milyen más változtatások szükségességét vonja maga után. Ráadásul a program méretének növekedésével egyre költségesebbé válnak azon regressziós tesztek végrehajtásai is, amik a program minőségéért felelősek. A hatásanalízis célja, hogy felderítse azon program pontokat, amelyeket a program egy adott pontjának megváltoztatása befolyásol valamilyen módon, illetve célja lehet az is, hogy az állandóan futó regressziós tesztelés lépéseit olyan módon redukálja, hogy az valóban csak a változtatás által érintett teszteket futtassa újra, ezzel is hatékonyabbá téve a tesztelési folyamatot.

Ebben a részben magasszintű nyelvek függőségi elemzését vettük célba. Attól függően, hogy milyen szintű nyelvi elemek között próbálunk függőségeket találni, más és más módszerek lehetségesek. A megváltozott programpont előrehaladó szeletének meghatározása egy természetes megoldás arra, hogy meghatározzuk egy változtatás által érintett utasítások halmazát [27]. A gyakorlatban azonban sokszor az olyan egyszerűbb és kevésbé költséges módszerek használatosak, amelyek egyszerűbb függőségi gráfokon dolgoznak, mint a szeletelés [3; 18; 44]. Ezt sokszor az is indokolja, hogy az esetek többségében nem szükséges utasítás szintű kapcsolatokat feltárnunk, elegendő csupán eljárás szintű, illetve csak osztály szintű kapcsolatok meghatározása.

Amennyiben csak a hívási gráf, illetve a viszonylag könnyen meghatározható hívási kapcsolatok segítségével közelítjük egy változtatás által érintett eljárások halmazát, akkor egy pontatlan közelítését kapjuk a valódi függéseknek [18]. Persze vannak olyan megközelítések, amik kombinálják az előbbi két megoldást, azaz ahol a hívási gráfon kapott eredményt pontosítják a programszeleteléssel [50].

Law és Rothermel által bevezetett `PathImpact` algoritmus függőség alapú dinamikus hatásanalízissel foglalkozik [43; 44], amely tömörített futási információkat generál a program instrumentált változatának végrehajtásával. A futási információk előre és hátrahaladó bejárásaival határozzák meg az adott eljárás előtt, illetve az eljárás után végrehajtott eljárások halmazait, amelyeket ezután az adott eljárás hatáshalmazainak tekintenek. Orso és társai a `CoverageImpact` algoritmusban [50] először meghatározzák azon lehetséges futtatásait a programnak, amelyek érintik a változtatott programpontot, majd meghatározzák ezen futtatások által lefedett eljárások halmazát, és végül a változtatott programpontból indított előrehaladó szelet segítségével kiválasztják ezen halmazok közül azokat, amelyeket a változtatás valóban befolyásol.

Apiwattanapong és társai összehasonlították az előbbi két eljárást a saját megvalósításukkal [3]. A gyakorlati összehasonlításukban adott programok különböző verzióit vették, amelyek teszteseteit felhasználva előállították az algoritmusok által igényelt dinamikus adatokat. Az egyes eljárásokat azok pontossága és hatékonysága szerint csoportosították. Ez alapján igazolták, hogy az általuk adott eljárás, amely az eljárások Execute After relációinak meghatározására épül – azaz azon programpontok feltérképezésére egy adott pont viszonylatában, amelyekre a vezérlés az adott pont végrehajtása után kerül –, legalább olyan pontos, mint a `PathImpact` algoritmus, és legalább olyan hatékony, mint a `ChangeImpact` analízis.

Felhasználva az előbb megismert Execute After relációkat, Beszédes és társai definiálták a `DynamicFunctionMetric` metrikát a program eljárásai között, amely segítségével egy pontosabb utat adnak az eljárás szintű hatáshalmazok meghatározására [11]. Gyakorlati eredményeik igazolják, hogyha egy adott eljárásra vesszük a tőle különböző DFC távolságra levő eljárások halmazait, akkor az a halmaz, amelynek elemei közelebb vannak az adott eljáráshoz, sokkal inkább fog olyan eljárásokat tartalmazni, amik valóban

függnek az adott eljárástól, mint az, ami távolabb van tőle. A metrikák használata a függőségek közelítésében nem új megoldás. Metrikák segítségével rangsorolhatjuk egy objektumorientált rendszer osztályait annak érdekében, hogy megjósoljuk, mely osztályok tartalmaznak legvalószínűbben hibás, vagy módosítandó kódrészleteket [19; 63]. Ezek a technikák azonban nem alkalmasak arra, hogy minden lehetséges függést megadjanak, csupán a legvalószínűbb függőségeket emelik ki.

Bináris programoknál a fő célunk az volt, hogy a függőségi analízis során előállított függőségi gráfok pontosításával a programszeletelés eredményét minél pontosabb formában adjuk meg. Ezzel ellentétben ebben a részben az alapvető célunk az, hogy minél jobban közelítsük a szeletelés standardnak vett eredményét olyan módszerrel, amely sokkal hatékonyabb és gyorsabb nagy szoftverrendszerek vizsgálatakor is, de ennek ellenére csak kevésbé pontatlan, mintha a programszeletelést alkalmaznánk. Így ez a módszer hatékonyan alkalmazható lehet olyan alkalmazásokban, mint a hatásanalízis, változás propagáció, tesztelés. Ráadásul célunk az is, hogy az általunk megadott függőségek halmaza teljes legyen, és az olyan függőségeket is képes legyen felfedezni, melyeket az egyszerűbb módszerek nem vesznek észre.

Az általunk bevezetett módszer alapját Apiwattanapong és társai által bevezetett Execute After reláció adja, aminek mintájára definiáljuk a Statikus Execute After és Statikus Execute Before relációkat. Előállításukhoz az elemzett programoknak egy olyan redukált gráf reprezentációját definiáljuk, amelynek a megfelelő bejárásai alkalmasak a keresett relációk meghatározására. A relációk meghatározására több lehetséges algoritmust is megadunk ebben a részben. Gyakorlati eredmények segítségével belátjuk, hogy az általunk bevezetett relációk valóban alkalmasak arra, hogy eljárás, illetve osztály szinten közelítsék a szeletelés eredményét. Szintén gyakorlati mérések segítségével pedig azt is igazoljuk, hogy csupán általános metrikák használatával csak közelíteni tudjuk az objektumorientált programok osztályai között fennálló függőségeket, mindet valóban nem tudjuk felderíteni.

7. fejezet

SEA és SEB relációk

„Egy találmány születésének pillanata sohasem véletlen; az emberek mindig csak azt találják fel, amire szükségük van, és azt mindig fel is találják."

— Szerb Antal

Ebben a fejezetben bevezetjük a Statikus Execute After és a Statikus Execute Before relációkat, amelyek a fejezet eredményeinek alapját képezik. Több algoritmust is adunk ezek meghatározására, felhasználásuktól függ, hogy melyik alkalmazása a célravezető.

7.1. SEA és SEB relációk definiálása

A kiinduló ötletet Appiwattanapong és társai által bevezetett *Execute After (EA)* reláció adta [3]. Ezt a relációt ők az eljárások dinamikus hatáshalmazainak meghatározására használták. A definíciójuknak megfelelően az f és g eljárások EA relációban állnak egymással akkor és csak akkor, ha g bármely részének lefutását megelőzi az f bármely részének futása a program meghatározott futtatásainak a halmazán.

A statikus megfelelője ennek a relációnak a *Static Execute After (SEA)* reláció. Azt mondhatjuk, hogy az f és a g eljárások akkor és csak akkor állnak egymással SEA relációban, ha elképzelhető hogy van olyan része a g eljárásnak, aminek lefutását megelőzi f bármely részének a végrehajtása[1]. Mint ahogyan a hátrahaladó szeletelésnek, úgy a SEA relációknak is definiálhatjuk a duálisát, vagyis a *Static Execute Before (SEB)* relációt. Az f és g eljárások SEB relációban állnak egymással akkor és csakis akkor, ha elképzelhető, hogy van olyan eseménye a g eljárásnak, ami az f eljárás valamely eseménye előtt fut le.

[1]Egy eljárásba való belépés, az eljárás elhagyása, illetve az eljárás utasításainak végrehajtásai mind az eljárás eseményeit alkotják.

Apiwattanapong és társai [3], valamint Beszédes és társai [11] alapján a formális definíciója a SEA relációnak a következő alakban adható meg:

$$\mathsf{SEA} = \mathsf{CALL} \cup \mathsf{SEQ} \cup \mathsf{RET}[\cup \mathsf{ID}],$$

ahol

$$\begin{aligned}
(f,g) \in \mathsf{CALL} &\iff f \text{ (tranzitíven) hívja } g\text{-t,} \\
(f,g) \in \mathsf{SEQ} &\iff \exists h : f \text{ (tranzitíven) visszatér} \\
&\quad\ h\text{-ba és utána } h \text{ (tranzitíven)} \\
&\quad\ \text{hívja } g\text{-t,} \\
(f,g) \in \mathsf{RET} &\iff f \text{ (tranzitíven) visszatér } g\text{-be,}
\end{aligned}$$

illetve még az egészet kiegészíthetjük az ID identikus relációval, ami opcionálisan része lehet a SEA-nak. Annak, hogy az ID relációt nem feltétlenül vesszük bele a SEA halmazba csupán az az oka, hogy bizonyos alkalmazásoknál is csak az érdekel minket, hogy egy adott eljárás hogyan viselkedik a többi eljárással.

Egyszerűen beláthatjuk, hogy az uniója a megadott relációknak valóban alkalmas arra, hogy megadja a SEA relációk halmazát, amennyiben átfogalmazzuk a fenti definíciót. Mondhatjuk, hogy $(f,g) \in \mathsf{SEA}$ akkor és csakis akkor, ha létezik olyan útvonal a program vezérlési folyam gráfján, ahol bármely f_{entry} esemény, azaz az f eljárásba való belépés megelőzi valamelyik g_{exit} eseményt, azaz a g eljárásból való visszatérést. Hogyha adott eljárásba való belépéseket, illetve azoknak megfelelő elhagyásait párokba állítjuk, három lehetséges esetet kapunk. Az útvonal az f és g eljárás esemény párjait a következő sorozatoknak megfelelően tartalmazhatja[2]:

- f_{entry}, g_{entry}, g_{exit}, f_{exit} esetben az f eljárás (tranzitíven) hívja a g eljárást.
- f_{entry}, f_{exit}, g_{entry}, g_{exit} esetben létezik olyan h eljárás, amely először (tranzitíven) hívja az f eljárást és utána (tranzitíven) hívja a g eljárást.
- g_{entry}, f_{entry}, f_{exit}, g_{exit} esetben az f eljárás (tranzitíven) visszatér a g eljárásba.

Mivel az f_{entry} mindig megelőzi az f_{exit} eseményt, így az identikus reláció szintén része lehet a SEA relációnak.

[2] f_{entry}, g_{entry}, f_{exit}, g_{exit} és g_{entry}, f_{entry}, g_{exit}, f_{exit} olyan szkenáriók, amelyek nem következhetnek be.

7.2. Az ICCFG gráf

Ahhoz, hogy meghatározhassuk egy program SEA relációinak a halmazát, egy erre megfelelő programreprezentációt kell előállítanunk. A hagyományos hívási gráf nem elegendő [54], hiszen ez semmit nem mond az egy eljáráson belül megvalósuló eljáráshívások sorrendjéről. Másfelől az interprocedurális vezérlési folyam gráf *(Interprocedural Control Flow Graph - ICFG)* [41] túl sok olyan információt tartalmaz, amely már nem köthető az eljárás hívásokhoz, így számunkra feleslegesek.

Először definiáljuk az intraprocedurális komponens vezérlési folyam gráfot *(Component Control Flow Graph - CCFG)*, ahol intraprocedurális szinten csak a hívási helyeket vesszük figyelembe. Minden CCFG gráf egy eljárást hivatott reprezentálni azzal, hogy tartalmaz egy belépési csomópontot, valamint több komponens csomópontot. Ezeket a komponens csomópontokat úgy kapjuk, hogy meghatározzuk az eljárás vezérlési folyam gráfjának az erősen összefüggő részgráfját, majd az egyes komponens csomópontokat a köztük levő vezérlési élekkel kötjük össze, ráadásul elhagyjuk azokat a csomópontokat, amelyek nem tartalmaznak hívási helyeket[3]. Az eljárások hívási pontokat tartalmazó összefüggő komponenseinek megadható egy tetszőleges topológikus rendezése is, ami rendezés fontos lesz a 7.6. ábrán bevezetett algoritmusnál.

Az interprocedurális komponens vezérlési folyam gráfban *(Interprocedural Component Control Flow Graph - ICCFG)*, amely segítségével meghatározzuk a SEA és SEB relációk halmazait, minden eljárás egy egyszerű CCFG gráffal adott, amely CCFG gráfokat a hívási élek kötik össze. Egy adott C komponensből hívási él indul az m eljárás belépési pontjába akkor és csak akkor, ha legalább egy olyan hívási helyet tartalmaz a C komponens, amely hívja az m eljárást.

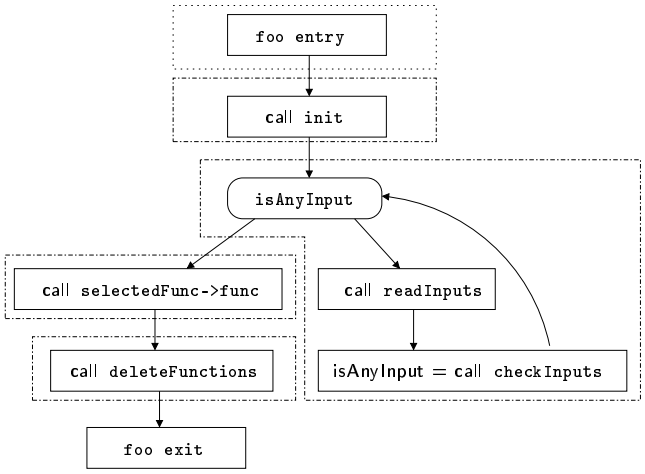
Az 7.1. ábrán bevezetett példa segítségével megpróbáljuk bemutatni, miként is épül fel az ICCFG gráf. A példa program célja egy olyan függvény futtatása, amelyet a felhasználó választ ki, illetve amely függvény bemenetét is a felhasználó adja meg. Amennyiben a felhasználó által megadott függvény nincs definiálva a program `init` függvénye által, úgy a felhasználónak egy új függvényt kell választania. Amennyiben a felhasználó által megadott bemenet nem érvényes, azaz jelen esetben nem konvertálható `double` típusúvá, akkor a bemenete a végrehajtandó függvénynek legyen 0.

A program `init` függvénye tehát inicializálja a program által végrehajtható függvények listáját. Nyilvánvaló, hogy ezt a listát tetszőlegesen kiegészíthetjük további `double` típusú adatokon értelmezett függvényekkel. A `readInputs` függvény beolvassa a felhasználó bemeneteit, míg a `checkInputs` függvény ellenőrzi, hogy a felhasználó által megadott függvény szerepel-e a program által végrehajtható függvények listáján. A program a megfelelően megadott függvényt függvény pointerek segítségével fogja

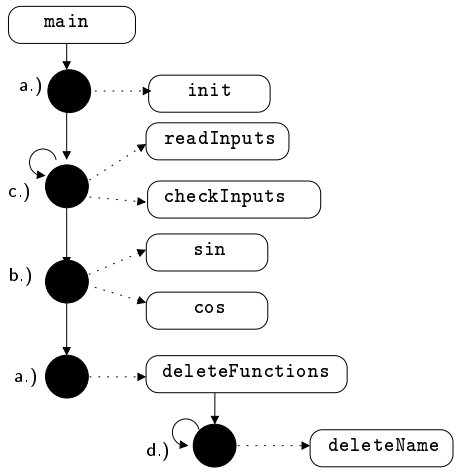
[3]Amennyiben két hívási hely kölcsönösen elérhetőek egymásból a vezérlési éleken haladva, úgy azok ugyanabba az erősen összefüggő komponensbe kerülnek, és így ugyanazzal a csomóponttal ábrázoltak a CCFG gráfban.

```
#include <iostream>
#include <math.h>
#include <string>
using namespace std;

typedef struct Func{
  string* name;
  double (*func)(double);
  struct Func* nextFunc;
} *Function;

void init (Function* list){
  Function tmp = (Function) malloc (sizeof(struc Func));
  tmp->name = new string("sin");
  tmp->func = sin;
  tmp->nextFunc = NULL;
  *list = tmp;
   ...
}

void deleteName(Function elem){
  if (elem->name)
    delete(elem->name);
}

void deleteFunctions (Function list){
  Function tmp=list;
  while (tmp){
    Function next = tmp->nextFunc;
    deleteName(tmp);
    free(tmp);
    tmp = next;
  }
}

void readInputs(string* inputFunction, double* input){
   cout << "What is the selected function?  ";
   cin >> *inputFunction;
   cout << "What is the argument?  ";
   string arg;
   cin >> arg;
   *input = atof(arg.c_str());
}

bool checkInputs(Function list, string input, Function* selected){
  Function tmp = list;
  while (tmp){
    if ((*tmp->name).compare(input) == 0)
      return true;
    tmp = tmp->nextFunc;
  }
  return false
}

int main(int argc, char *argv[]){
  Function functionList = NULL;
  init(&functionList);
  string inputFunc;
  Function selectedFunc = NULL;
  double input = 0.0;
  bool isAnyInput = false;
  while (!isAnyInput){
    readInputs(&inputFunc, &input);
    isAnyInput = checkInputs(functionList, inputFunc, &selectedFunc);
  }
  cout <<inputFunc <<"("<<input<<")=" <<selectedFunc->func(input) << endl;
  deleteFunctions(functionList);
}
```

7.1. ábra. Példa program



7.2. ábra. A CCFG gráfja a 7.1. ábra `main` függvényének



7.3. ábra. Az ICCFG gráfja a 7.1. ábra példájának.

végrehajtani. A program által lefoglalt erőforrások felszabadításáért a program végén végrehajtott `deleteFunctions` és `deleteName` függvények felelősek.

A 7.2. ábra mutatja a vezérlési folyam gráfját a példa `main` függvényének, amely a legkomplexebb függvénye a programnak. Az erősen összefüggő komponenseit a vezérlési folyam gráfnak szaggatott vonalak határolják. A CCFG gráfot a hívási helyeket tartalmazó komponensek, illetve a köztük futó vezérlési élek által határozhatjuk meg. Ezek a komponensek lesznek a 7.3. ábra fekete csomópontjai. Az egyszerűség kedvéért az ábra csak a belépési csomópontját ábrázolja azoknak a függvényeknek, amik legfeljebb csak könyvtári függvényhívást tartalmaznak.

Négy féle komponens létezik.

a.) A komponens csak egyetlen hívási helyet tartalmaz, amelynek célpontja egyértelműen meghatározott.

b.) A komponens egyetlen hívási helyet tartalmaz, de több különböző célpontja lehet ennek a hívásnak. Ez a szituáció akkor fordul elő, amikor fordítási időben nem lehet eldönteni a hívás helyét egyértelműen. Az ilyen eseteket okozhatja függvény pointerek használata, illetve objektumorientált nyelveknél virtuális metódushívások. Ezen a ponton a gráf reprezentáció igen konzervatív lehet. Minél pontosabb hívási gráfját használjuk a programnak, annál pontosabb lesz a mi reprezentációnk is. A hívási gráfot pontosíthatjuk egy alkalmas pointer analízis segítségével, illetve objektumorientált programok esetében Rapid Type Analysis [4] analízis segítségével.

c.) A komponens egy vagy több hívási helyet tartalmaz, amelyek ugyanabban a ciklusban vannak, és amelyeknek több célpontja lehet. Ez az eset megegyezhet a b pontban tárgyalt esettel, attól eltekintve, hogy most a hívás egy ciklusba van beágyazva. Esetünkben a hívás(ok) valamennyi lehetséges célpontja SEA relációba kerül egymással, hiszen bármelyik célpont lefuthat bármelyik célpont végrehajtása előtt.

d.) A komponens egy hívási helyet tartalmaz, egyetlen célponttal, de a hívási hely egy ciklus része. Ebben az esetben elképzelhető, hogy az eljárást többször futtatjuk egymás után, így a hívott eljárás önmagával még akkor is SEA relációban áll, ha egyébként az ID relációt nem tekintjük a SEA részének.

Az általunk meghatározott CCFG programreprezentációhoz hasonló Badri és társai által használt reprezentáció [5]. Az ő céljuk szintén függőségekre épülő statikus hatáshalmazok megadása. Módszerük kiindulási pontja hasonló az általunk bemutatott eljáráshoz. A statikus hatáshalmazok meghatározásához veszik a programot reprezentáló *control call* gráfot, amely gyakorlatilag a vezérlési folyam gráf, és hívási gráfból áll elő úgy, hogy azokat a csomópontokat elhagyják, amelyek nem befolyásolják az eljáráshívások végrehajtását. A különbség az ő és a mi reprezentációnkban az, hogy mi a feltételeket, elágazásokat reprezentáló csomópontokat sem tartjuk meg, illetve a kapott csomópontokat erősen összefüggő komponenseik által össze is vonjuk, ezzel is redukálva a reprezentáció méretét. Badriék algoritmusa minden eljárásra meghatározza a bennük lévő eljáráshívások tömör sorozatát, különböző jelöléseket bevezetve az iterációban, illetve feltételek különböző ágában részt vevő hívásokra.

program Seb(P, f)
input: $P-$ a program ICCFG reprezentációja
$f-$ P-beli eljárás
output: $S-$ olyan halmaza eljárásoknak, amelyek SEB relációban állnak f-fel
begin
1 Ürítsük ki az S halmazt!
2 Minden P-beli eljárás belépési pontja legyen színtelen!
3 Az f eljárás belépési pontját színezzük szürkére!

4 Járjuk be P-t az f belépési pontjából az éleken hátrafelé haladva úgy, hogy minden érintett csomópont valamennyi bejövő élét pontosan egyszer érintsük!
5 Ha egy c komponens elérhető egy másik komponensből (nem belépési pontból), akkor a c által hívható eljárások belépési pontjait színezzük szürkére!
6 Ha a bejárás során elérjük egy e eljárás belépési pontját, akkor tegyük be e-t S-be!

7 Amíg van szürke belépési pont, addig
8 legyen e egy szürke belépési pont,
9 színezzük e-t feketére,
10 és minden e által hívható színtelen g eljárás belépési pontját színezzük szürkére és tegyük g-t az S-be!
end

7.4. ábra. SEB relációk meghatározása gráfelérési problémaként

7.3. Algoritmusok a SEA és SEB relációk meghatározására

A [34] cikkünkben bevezettünk egy algoritmust, amely egy tetszőleges eljárásra megadja azon eljárások halmazát, amelyek SEB relációban állnak az adott eljárással. Ezen algoritmus párját, amely egy adott eljárás SEA halmazát határozza meg, a [33] cikkben ismertettük. Bár ha adott valamely relációt meghatározó algoritmusunk, akkor a másik reláció meghatározása egyszerűen adódna az élek ellenkező irányba való bejárásával, mégsem ezt tesszük. Ennek oka, hogy sokkal tömörebb reprezentációval dolgozhatunk, ha a hívási éleket csupán a hívó oldalon jegyezzük fel. Így viszont annak meghatározása, hogy egy adott eljárásból mely eljárásokba térhetünk vissza, sokkal költségesebb feladat

```
program Sea(P, f)
input: P – a program ICCFG reprezentációja
       f – P-beli eljárás
output: S – olyan halmaza eljárásoknak, amelyek SEA relációban állnak f-fel
begin
1   Ürítsük ki az S halmazt!
2   A P komponensei legyenek színtelenek!
3   Színezzük f minden komponensét szürkére!
4   Járjuk be P-t az f belépési pontjából az éleken hátrafelé haladva úgy,
    hogy minden érintett csomópont valamennyi bejövő élét pontosan egyszer
    érintsük!
5       Ha egy c komponenst elérünk egy belépési pontból (azaz nem
        egy másik komponensből), színezzük c minden rákövetkező
        komponensét szürkére!
6       Ha egy e eljárás belépési pontját elérjük a bejárás által, akkor
        tegyük be e-t S-be!

7   Amíg van szürke komponens, addig
8       legyen c egy tetszőleges szürke komponens,
9       színezzük c-t feketére,
10      színezzük c színtelen rákövetkezőit szürkére,
11      minden g eljárásra, amit hívhat a c
12          tegyük be g-t S-be, és a
13          színtelen komponenseit g-nek színezzük szürkére!
end
```

7.5. ábra. SEA relációk meghatározása gráfelérési problémaként

lenne.

A 7.4. ábra algoritmusa a SEB, míg a 7.5. ábra algoritmusa a SEA halmazát határozza meg egy megadott eljárásnak. Mind a két algoritmus két menetben bejárja a programot reprezentáló ICCFG gráfot, a bejárás közben színezi a csomópontokat, és a színezésnek megfelelően végül meghatározzák a kívánt halmazokat.

A [12] cikkünkben olyan algoritmust adtunk a SEA, és egyben SEB relációk megadására, amely egy időben valamennyi eljárás SEA, illetve SEB relációit meghatározza. A [33] cikkben pontosítottuk az eredeti algoritmust azzal a megfigyeléssel, hogy a gráfunkat, illetve az algoritmust nemcsak azzal tudjuk hatékonyabbá tenni, ha az egyes eljárások vezérlési folyam gráfjai helyett azok erősen összefüggő komponenseit használjuk,

program SeaSeb(P)

input: $P-$ a P program ICCFG gráfja

output: $SEA-$ az összes SEA reláció párja a programnak
$SEB-$ az összes SEB reláció párja a programnak

begin

1 Határozzuk meg a hívási gráf összefüggő komponenseinek topológikusan rendezett $SCC\_CallGraph$ listáját!
2 Minden cc elemére az $SCC\_CallGraph$ listának utolsótól az elsőig
3 A cc rákövetkező komponens minden cc_next rákövetkező komponensére
4 cc CALL relációi közé szúrjuk be cc_next-et, illetve a cc_next-tel CALL relációban álló komponenseket;
5 minden $(x, y) \in$ CALL komponenspárra (y, x) legyen RET eleme!

6 Minden m eljárására a programnak
7 legyen $topOrder$ az m komponenseinek topológikusan rendezett listája;
8 minden c komponensére a $topOrder$ listának elsőtől az utolsóig
9 $prev[c]$ legyen a c-t megelőző p komponensek $prev[p]$ halmazainak az uniója;
10 ha c egy ciklus eleme, akkor
11 $prev[c]$-hez vegyük hozzá a c-vel CALL relációban álló komponenseket;

12 a SEQ relációkat bővítsük $prev[c]$ és a c-vel CALL relációban álló komponensek halmazának Descartes-szorzatával;
13 ha c nincs ciklusban
14 $prev[c]$-hez vegyük hozzá a c-vel CALL relációban álló komponenseket!

15 A SEA reláció legyen a CALL, RET és SEQ relációk uniója!
16 Minden $(x, y) \in$ SEA párra (y, x) legyen SEB eleme!

7.6. ábra. SEA és SEB relációk meghatározása a programban

de azzal is növelhetjük a hatékonyságot, ha a hívási gráfnak is meghatározzuk az erősen összefüggő komponenseit. A hívási gráf ugyanazon összefüggő komponensébe tartozó eljárások SEA és SEB halmazaira a következő állítást fogalmazhatjuk meg:

Lemma: *Legyen $SCC\_CallGraph$ a hívási gráf erősen összefüggő komponensekből álló részgráfja. Legyen F és G két komponense az $SCC\_CallGraph$-nak, ahol az f és g eljárásokra igaz, hogy $f \in F$ és $g \in G$. Ha $(f, g) \in SEA$, akkor F és G minden elempárjára igaz ugyanez. Hasonlóan ha $(f, g) \in SEB$, akkor F és G minden elempárjára igaz az, hogy azok SEB relációban állnak egymással.*

Bizonyítás: Egy komponensen belüli eljárások tranzitív hívási relációban állnak egymással. Minden $f'(\neq f) \in F$-re létezik tranzitív CALL és RET reláció f' és f között, és hasonlóan igaz ez minden $g'(\neq g) \in G$ és a g eljárásokra. Ha f és g CALL, RET vagy SEQ relációban állnak egymással, akkor minden $f' \in F$ és $g' \in G$ pár is CALL, RET vagy SEQ relációban lesznek egymással. Az, hogy a CALL és a RET relációk fennállnak ezen relációk tranzitivitásából következik, míg a SEQ reláció létezése annak definíciójából. □

A 7.6. ábrán bemutatott algoritmus valamennyi eljárás SEA és SEB halmazait egyszerre határozza meg. Első lépésként az algoritmus meghatározza a hívási gráf erősen összefüggő komponenseit. A 2-5. sorok felelősek a CALL és RET relációk meghatározásáért. Ezek a relációk mind a SEA, mind a SEB halmazok részét képezik. Minden komponensre megadjuk azon komponensek halmazát, amelyek elemei elérhetőek az adott komponens eljárásaiból hívási élek által, illetve amelyek eljárásai hívhatják a vizsgált komponens eljárásait. A 6-14. sorok határozzák meg azon SEQ relációkat, amelyek egy eljárás különböző hívási helyei között realizálódnak. Mivel az eljárások komponenseit azok topológikus sorrendjében dolgozzuk fel, ez garantálja, hogy amikor a c komponenst vizsgáljuk, akkor annak minden p megelőző komponensét már vizsgáltuk, és így az az információ, hogy a p végrehajtása előtt mely eljárásokat hívhattuk, valamint az, hogy a p mely eljárásokat hívhatja a végrehajtása közben, már adott információk. Így amikor a 14. sorban a c komponens $prev$ halmazát kiegészítjük, az tartalmazni fogja az összes olyan eljárást, amelyeket a c végrehajtása előtt meghívhatunk az adott m eljárás valamely futtatásakor. A 15-16. sorokban figyelembe kell vennünk azt, hogy a CALL, RET, és SEQ halmazok a hívási gráf erősen összefüggő komponensei között adják meg a nevezett relációt. Szintén ennél a pontnál biztosíthatjuk azt a feltételt, hogy mind a SEA, mind a SEB reflexív relációk legyenek.

7.4. Az algoritmusok összehasonlítása

Érdekes kérdés lehet, hogy adott alkalmazás esetében melyik algoritmus használata a célravezető a kívánt relációk meghatározására. Elképzelhető olyan eset, amikor nem szükséges számunkra, hogy valamennyi eljárás SEA és SEB halmazait meghatározzuk,

csupán néhányét. Ilyenkor egyértelmű lehet, hogy a 7.4. és 7.5. ábrákon megadott algoritmusok használata lehet a hatékonyabb választás. A valóságban azonban egyszerre általában nem csak egy-egy eljárás SEA és SEB relációi érdekelnek majd bennünket. Habár számtalan tényező befolyásolja a komplexitását az algoritmusainknak, és nehéz egy pontos határvonalat adni, hogy az egyes esetekben melyiket célszerű használni, azért megpróbáljuk egy kicsit összehasonlítani őket. Először vegyük sorba az algoritmusok azon lépéseit, amelyek döntően meghatározzák a költségüket.

- A 7.4. ábra algoritmusa

  a) 1-3. sorok: inicializálás

  b) 4-6. sorok: gráfbejárás a bejövő éleken haladva, amely bejárás költsége arányos a gráf magasságával. Ez a lépés meghatározza a RET relációkat, és a színezésekkel előkészítik a következő bejárást.

  c) 7-11. sorok: a második gráfbejárást megvalósító utasítások. Azon csomópontok, amiket olyan eljárásokból érünk el direkt, vagy indirekt módon ezen bejárás során, amik belépési pontját az 5. sorban színeztük szürkére, az f eljárással SEQ relációban állnak. Ez a bejárás adja meg az f eljárás CALL relációit is azon eljárások összegyűjtésével, amelyeket amiatt érünk el, hogy a 3. sorban szürkére festettük az f belépési pontját.

- A 7.5. ábra algoritmusa

  a) 1-3. sorok: inicializálás

  b) 4-6. sorok: gráfbejárás a bejövő éleken haladva, amely bejárás költsége arányos a gráf magasságával. Ez a lépés meghatározza a RET relációkat, és a színezésekkel előkészítik a következő bejárást.

  c) 7-13. sorok: a második gráfbejárást megvalósító utasítások. Az f eljárás azokkal az eljárásokkal lesz SEQ relációban, amelyeket ennél a bejárásnál olyan csomópontokból érünk el direkt, vagy indirekt módon, amelyeket az 5. sorban színeztünk szürkére. Továbbá ez az eljárás adja meg az f CALL relációit is azon eljárások összegyűjtésével, amelyeket a 3. sor színezése miatt érünk el.

- A 7.6. ábra algoritmusa

  a) 1-5. sorok: CALL és RET relációk meghatározása

  b) 6-16. sorok: az összes eljárás összes komponensének egyszeri vizsgálata, amely során előállítjuk a Descartes-szorzatát azon eljárások halmazának, amelyeket az adott komponens végrehajtása előtt érintettünk az adott eljárás

végrehajtása közben, illetve azon eljárások halmazának, amely összegyűjti az adott komponens által direkt vagy indirekt módon hívható eljárásokat. Ez a lépés tehát meghatározza a SEQ relációk halmazát, amelyek az adott m eljárás közvetítésével valósulnak meg.

A legnagyobb különbség az első kettő és a harmadik algoritmusok között tehát az, hogy míg ez utóbbi valamennyi SEA és SEB relációt meghatározza, addig az első kettő algoritmus csak az egyik relációfajta meghatározására képesek, ráadásul azt is úgy, hogy egy meghatározott eljárásra határozzák meg az adott reláció halmazt. Így ha az első kettő algoritmus költségét össze szeretnénk hasonlítani a harmadik algoritmus költségével, úgy ezek eredeti költségét meg kell szoroznunk az eljárások számával. Szemben a harmadik algoritmussal, az első kettőnek többször be kell járnia a gráfot, amennyiben több eljárás SEA, illetve SEB halmazait szeretnénk általuk megadni. Mindazonáltal még ez a többszörös bejárás is kedvezőbb lehet sok esetben, mint a Descartes-szorzat számításai a harmadik algoritmusnak.

Az algoritmusok legrosszabb esetre vett komplexitásai a következők: legyen n az eljárások száma, k és e a maximális komponens és él számai az egyes eljárásoknak, és legyen m a maximális eljárás hívások száma, ami egy komponensből indul.

Az első két algoritmus kétszer járja be a gráfot annak érdekében, hogy a meghatározott metódus kívánt relációit összegyűjtse. Mindkét bejárás lineáris a gráf méretének függvényében, így az eljárások legrosszabb eset komplexitásai $O(n \cdot e + n \cdot k \cdot m)$, ha mind az n eljárás SEA és SEB relációit meg szeretnénk határozni általuk. A harmadik algoritmus először meghatározza az egyes eljárások tranzitív hívásait, meghatározza az eljárások komponenseinek topológikus sorrendjét, és a komponenseket végigjárva különböző halmaz-műveletek segítségével állítja elő a kívánt relációk halmazait. Megfelelő adatstruktúrákat használva ezt szintén $O(n \cdot e + n \cdot k \cdot m)$ idő alatt teszi meg legrosszabb esetben.

Ezek a komplexitások nemcsak hogy egymással versenyeznek, de tulajdonképpen megegyezik az SDG-alapú statikus szeletelésnek a számítási komplexitásával [32], azzal a nem elhanyagolható különbséggel, hogy a szeletelésnél használt SDG gráf sokkal nagyobb (nagyságrendekkel több élt és csúcspontot tartalmaz), mint az általunk felépített ICCFG gráf. Ahhoz, hogy előállítsuk az SDG-t, illetve az ICCFG-t, a kiindulási alapunk lehet egy interprocedurális vezérlési folyam gráf (ICFG). Míg az ICCFG ebből csomópontok törlésével, illetve mélységi bejárással előállított erősen összefüggő komponensek által adódik, addig az SDG felépítése adat függőségek, vezérlés függőségek meghatározását igénylik, amelyek önmagukban is egy-egy komplex számítás eredményeként adódnak. Vagyis összességében az ICCFG gráf meghatározásának komplexitása szignifikánsan jobb, mint az SDG meghatározása.

Visszatérve a saját algoritmusaink összehasonlításához, mivel a legrosszabb eset számítás nem hozott eltérést az algoritmusok között, egy gyakorlati mérés segítségével

mutatjuk meg a különbségeket. Ehhez a méréshez a gcc fordító forráskódjának elemzését választottuk, mivel ez egy megfelelően nagy program már ahhoz, hogy segítségével mérni tudjuk az algoritmusok közötti különbséget. Ahhoz, hogy az eredményeink összehasonlíthatóak legyenek, ugyanazt a keretrendszert és adat reprezentációt alkalmaztuk ahhoz, hogy a felismert relációkat összegyűjtsük. Mérésünk során csak SEA relációkat számítottunk úgy, hogy valamennyi eljárásra lefuttattuk a 7.5. ábra algoritmusát, majd lefuttattuk a 7.6. ábra algoritmusát. A teljes futási ideje a 7.5. ábra algoritmusának négyszerese volt a 7.6. ábra algoritmus futásidejének, de még úgy is kétszeres volt a futásideje, ha az eredeti megvalósításunkkal futtattuk az utóbbi algoritmust, ahogy azt a [12] cikkben publikáltuk. Emlékeztetőül ebben az esetben ahelyett, hogy egyben kezelnénk azon eljárások SEA és SEB halmazait, amelyek a hívási gráf ugyanazon erősen összefüggő komponensében vannak, minden eljárás esetén külön tartjuk nyilván a kapcsolatokat, amelyek sokkal nagyobb halmazokhoz, illetve nagyobb halmazokon végzett műveletekhez vezetnek.

Természetesen még ez a gyakorlati mérési eredmény sem ad elegendő támpontot ahhoz, hogy melyik algoritmus használata a célravezető egy meghatározott szituációban. Ha a meghatározott relációk memóriában tartása nem probléma, akkor kijelenthetjük, hogy minél több eljárás SEA, illetve SEB halmazainak a meghatározásában vagyunk érdekeltek, annál valószínűbb, hogy a 7.6. ábra algoritmusát célszerű használnunk.

7.5. Eredmények összegzése

Ebben a fejezetben bevezettük a Statikus Execute After és a Statikus Execute Before relációkat, amelyekkel első megközelítésben az elemzett program eljárásai között fennálló függőségeket szeretnénk közelíteni. Definiáltuk a programnak azt a reprezentációját, az ICCFG gráfot, amely megfelelő bejárásával alkalmas arra, hogy megadja ezeket a relációkat. A fejezetben lehetséges algoritmusokat adtunk arra vonatkozóan, miként lehet ezeket a bejárásokat megvalósítani. A SEA, illetve SEB relációk, valamint az ICCFG definiálása a szerzőtársakkal közös, oszthatatlan eredmények. Az algoritmusok, illetve az algoritmusokhoz köthető mérési eredmények a szerző önálló eredményei.

8. fejezet

SEA és SEB relációk a hatásanalízisben

„Jósolhatóság: egy pillangó szárnylebbentése Brazíliában okozhat-e tornádót Texasban?"

— Edward Lorenz

Ebben a fejezetben gyakorlati mérések által megpróbáljuk belátni, hogy a SEA és SEB relációk meghatározásával egy olyan biztonságosan használható eszközt adhatunk meg eljárás szintű hatásanalízishez, amely lehetséges alternatívája lehet a sokkal több erőforrást igénylő szeletelésnek ezen a területen. Természetesen érdekes lenne pontos matematikai formulákkal megadni, hogy a SEA és SEB relációk hogyan viszonyulnak a statikus szelet méretekhez, de nyilvánvalóan ilyen összehasonlítást nem lehet adni, hiszen a statikus szeletek önmagukban is különbözőek lehetnek attól függően, hogy a használt szeletelő eszköz az egyes függőségeket, amelyek alapján a szeleteket meghatározza, milyen pontossággal határozza meg.

8.1. Fedettség és pontosság értékek definiálása

Méréseink során a meghatározott SEA, illetve SEB relációkat egy erre alkalmas szeletelő eszköz eredményeivel vetettük össze. A SEA relációkat nyilván az előrehaladó szeletekkel, míg a SEB relációkat a hátrahaladó szeletekkel lehet összehasonlítani. Mivel a SEA és SEB relációkat eljárás szinten definiáltuk, az összehasonlításhoz a szeletelés eredményeit is eljárás szintre kell kiterjesztenünk. Egy adott eljárás szelethalmazát azon eljárások alkotják, amelyeknek van olyan utasítása, amely eleme egy olyan programszeletnek, amit a vizsgált eljárás egy tetszőleges kritériumából indítottunk.

Az összehasonlítás alapjául a pontosság ($precision = \frac{TP}{TP+FP}$) és fedettség ($recall = \frac{TP}{TP+FN}$) értékek szolgálnak, ahol

- az igaz pozitív példák (TP) halmaza azoknak az eljárásoknak a halmaza, amelyeket mind a szeletelő, mind a SEA (SEB) relációba állít a vizsgált eljárással,

- a hamis negatív példák (FN) halmaza azoknak az eljárásoknak a halmaza, amelyek bár a szeletelés szerint relációban vannak a vizsgált eljárással, a SEA (SEB) szerint nincsenek,

- a hamis pozitív példák (FP) halmaza azoknak az eljárásoknak a halmaza, amelyek a SEA (SEB) szerint relációban állnak egymással, a szeletelés szerint viszont nincsenek,

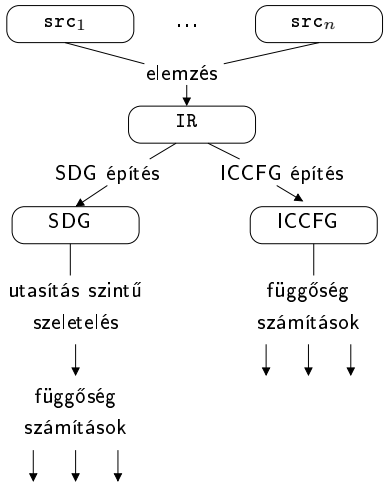
- az igaz hamis példák (TN) halmaza pedig azoknak az eljárásoknak a halmaza, amelyek sem a szeletelés, sem a SEA (SEB) szerint nincsenek relációban egymással.

Vagyis a pontosság értékek megadják, hogy a szeletelés és SEA (SEB) által is relációba állított eljárás párok száma hány százaléka a szeletelés által meghatározott relációpárok számának, illetve a fedettség értékek azt adják meg, hogy a SEA (SEB) által beazonosított kapcsolatoknak mekkora része valódi kapcsolat a szeletelés szerint.

A program egy tetszőleges kritériumából indított programszeletet megkaphatjuk a rendszerfüggőségi gráf éleinek megfelelő bejárásával [32]. Az érintett élek a vezérlés- és adatfüggőségi, valamint a paraméter élek, illetve az eljárások formális bejövő és kimenő paraméterei közötti összegző élek. Valamennyi függőség akkor jön létre, ha két érintett pont között van vezérlési élek által meghatározott út, így a szeletelés által behozott relációk eljárás szintre való kiterjesztése a SEA (SEB) relációk által meghatározott gráf valódi részhalmaza kell hogy legyen. Ez pedig 100%-os fedettség értéket feltételez minden esetben.

Ez a feltevés azonban nem teljesült akkor, amikor SEA halmazokat összevetettük a CodeSurfer programszeletelő eszköz előrehaladó szeletei által meghatározott halmazokkal. Kivizsgáltuk az okát ennek a jelenségnek, amely vizsgálat egy tanulságos programhibáját tárta fel a CodeSurfer 2.1p1 verziójának. A hibát és annak okait a függelék A részében ismertetjük. A hibának köszönhetően a gyakorlati mérésekben így csak a SEB halmazoknak és a hátrahaladó szeletek által meghatározott kapcsolatoknak a viszonyát vizsgáljuk[1].

[1] Amennyiben a dualitás teljesülne az előrehaladó és hátrahaladó szeletek között is, nemcsak a SEA és SEB relációk között, úgy az elemzett program valamennyi kritériumára végzett mérések átlagosan ugyanakkora pontosság és fedettség értéket adnának mindkét irányban.



8.1. ábra. Gyakorlati mérések megvalósításának architektúrája

8.2. Gyakorlati mérések megvalósítása

Ahhoz, hogy az algoritmusunk által kiszámított SEA és SEB halmazokat össze tudjuk hasonlítani a CodeSurfer programszeletelő eszköz [29] szeletelési eredményeivel, a CodeSurfer API-jának segítségével állítottuk elő az ICCFG gráfot. Természetesen a módszer független a CodeSurfer részletes számításaitól, és elemzésétől, csupán az elemzett programra meghatározott hívási gráf és intraprocedurális vezérlési folyam információk kinyerése fontos számunkra, amit nyilvánvalóan más, egyszerűbb eszközök segítségével is meghatározhatnánk. A 8.1. ábra ismerteti a szeleteléshez, illetve a SEA (SEB) relációk meghatározásához szükséges gráf reprezentációk előállításának lépéseit.

A CodeSurfer lehetőséget ad különböző reprezentációk előállítására attól függően, hogy a felhasználó milyen beállításokkal végzi a program elemzését. Egy közös frontendet használva, amely az adott program elemzéséért felelős, előáll egy köztes reprezentációja a programnak, amely azonban már némileg különböző lehet attól függően, hogy mire akarjuk felhasználni az elemzés eredményét. A szeleteléshez és a SEA (SEB) számításhoz különböző beállításokkal indítottuk az elemzést azért, hogy minél optimálisabb teljesítményt érjünk el az egyes megvalósításoknál. A 8.1. táblázat összefoglalja a különböző beállítások paramétereit. Mivel a SEA (SEB) relációk meghatározásához lényegesen kevesebb információ szükséges, mint a szeleteléshez, így a legtöbb költséges számítási részt kikapcsolhatjuk. Ahhoz, hogy egy programszeletet előállíthassunk a program egy meghatározott pontjából indulva, az adat és vezérlés függőségek mellett minden eljárásra rögzítenünk kell, hogy azok mely globális változókat írják, olvassák, ráadásul meg kell határoznunk az összegző éleket is, amelyek az egyes eljárások formális bejövő és kimenő paraméterei között teremtenek kapcsolatot. Az ICCFG előállításához ezzel ellentétben elegendő megadni csak a hívási gráfot, és elegendő kigyűjteni az egyes hívási helyeket a köztük levő vezérlési folyam információval. Meghatározva a hívási helyeket

Preset of CodeSurfer	SDG	ICCFG
-control-dependence	igen	nem
-data-dependence	igen	nem
-compute-gmod	igen	nem
-compute-summaries	igen	nem
-cfg-edges	nem	igen, mindkét irányba
-basic-blocks	nem	igen

8.1. táblázat. A CodeSurfer felhasználástól függő elemzési beállításai

reprezentáló csomópontok erősen összefüggő komponenseit, és ezen komponenseknek a topológikus sorrendjét, valamint eltárolva azt az információt, hogy az egyes hívási helyeknél mely eljárásokat hívhatjuk, egy egyszerű konverzióval előáll az ICCFG a program köztes reprezentációjából.

8.3. Elemzett programok bemutatása

Ahhoz, hogy a módszerünk pontosságát igazoljuk, Binkley and Harman által összegyűjtött teszteket használtuk [15], bár néhány esetben eltértünk az általuk elemzett program verzió számától. A 8.2. táblázat felsorolja az elemzett programokat, azok néhány alap adatával. A TL (total lines) érték az elemzett program sorainak számát, míg az LCode (Logical line) érték a CodeSurfer szerint ténylegesen soroknak tekinthető sorok számát adja meg.

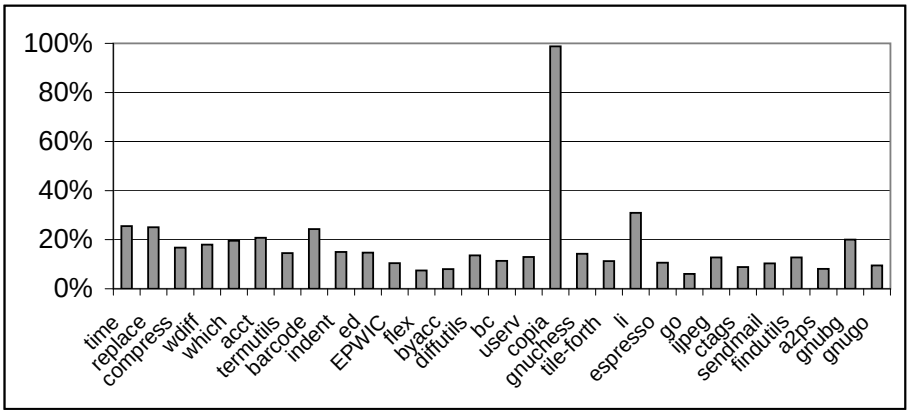
Annak érdekében, hogy a két megközelítés, azaz a SEA (SEB) számítás és a szeletelés különbségeit tár és számítási idő szerint is össze tudjuk hasonlítani, vettünk pár nagyobb C/C++ nyílt forráskódú programot is, amelyek alap jellemzőit a 8.3. táblázat foglalja össze.

8.2. táblázat. C nyelvű teszt programok a pontosság szemléltetéséhez

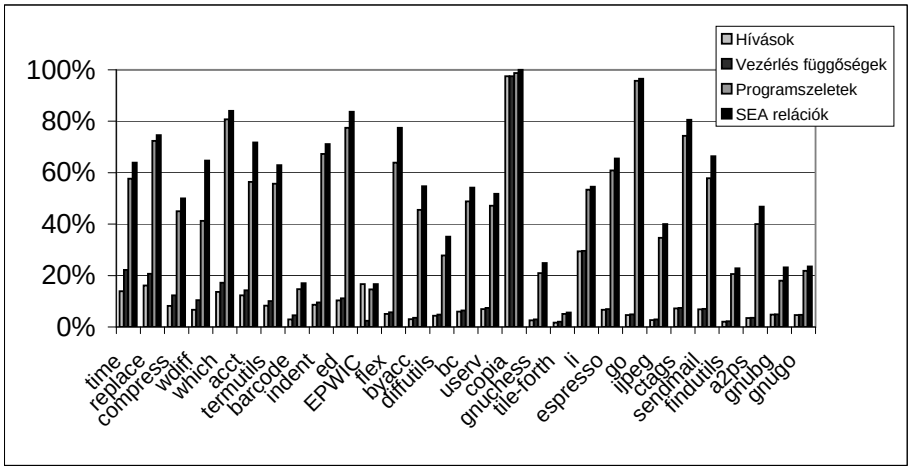
Programok	Eljárások száma	TL	LCode
time v1.7	12	1 314	757
replace v?	21	563	512
compress v?	24	1 937	1 335
wdiff v0.5	27	1 862	1 080
which v2.17	28	1 989	1 246
acct v6.3	50	3 510	1 996
termutils v2.0	57	3 685	2 518
barcode v0.98	62	3 885	2 331
indent v2.29	111	11 539	7 582
ed v0.8	120	3 052	2 267
EPWIC v?	149	9 597	5 249
flex v2.4.7	152	14 184	9 134
byacc v1.9	178	3 553	2 737
diffutils v2.8	192	15 022	9 735
bc v1.06	204	7 794	5 290
userv v0.95.0	239	7 909	6 016
copia v?	242	1 168	1 085
gnuchess v5.07	261	16 533	11 045
tile-forth v2.1	286	5 730	3 549
li v?	357	7 597	4 793
espresso v?	361	22 050	21 780
go v?	372	29 629	22 118
ijpeg v?	467	28 185	15 253
ctags v5.0	518	13 750	10 018
sendmail v8.14	548	123 965	77 950
findutils v4.2.31	608	41 661	27 261
a2ps v4.13	902	54 954	33 573
gnubg v1.2	192	6 705	4 330
gnugo v3.6	2 188	151 376	110 631

8.3. táblázat. C/C++ programok a hatékonyság méréséhez

Program	Eljárások száma	TL	LCode
valgrind 3.3.0 (C)	5 318	228 763	141 631
gdb 6.7.1 (C)	8 095	473 793	303 552
gcc 4.0 (C)	16 108	1 052 353	725 620
mozilla 1.6 (C++)	83 432	2 382 459	1 414 946



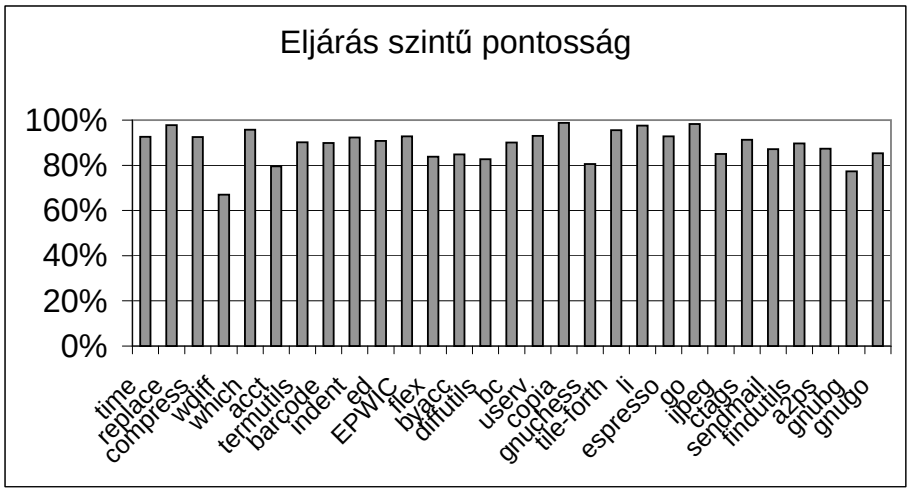
8.2. ábra. Fedettség értékek a hívási gráf által meghatározott hatáshalmazok esetében



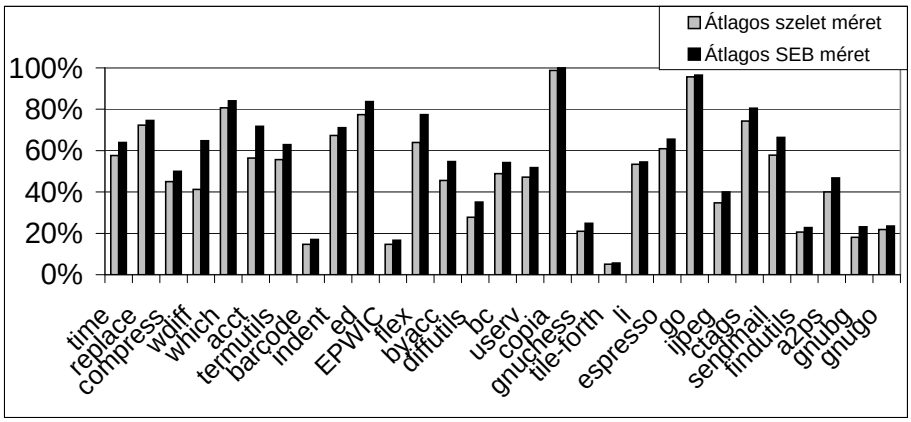
8.3. ábra. Hívási kapcsolatok, vezérlés függőségek, előrehaladó szelet méretek és SEA kapcsolatok számának összehasonlítása

8.4. Gyakorlati mérések kiértékelése

A 8.2. ábra azt mutatja, hogy önmagában a hívási gráf nem elegendő arra, hogy meghatározzuk egy adott eljárásra azon eljárások halmazát, amelyekre az hatással van. A fedettség értékek alacsonyak ebben az esetben, így láthatjuk, hogy a hívási gráf által meghatározott hatáshalmazok nagyon rosszul közelítik a valódi hatáshalmazokat. A 8.3. ábrán az egyes eljárásokhoz tartozó átlagos hívási kapcsolatok, a vezérlés függőségek, az előrehaladó szelet méretek és a SEA kapcsolatok számai kerülnek összehasonlításra. A vezérlés függőségek egy kicsit pontosabbak a hívási kapcsolatoknál, hiszen sokszor egy adott eljárás végrehajtásának a függvénye, hogy a vezérlés végrehajtása hol folytatódik az eljárás végrehajtása után, és így két eljárás között még akkor is lehet függőség, ha a két eljárás között hívási kapcsolat egyébként nincs. Mindazonáltal a vezérlés függőségek önmagukban még szintén nem elegendőek arra az ábrán feltüntetett adatok alapján, hogy segítségükkel meggyőzően közelíteni tudjuk egy adott eljárás lehetséges hatásainak



8.4. ábra. A SEB relációk pontossága a hátrahaladó szeletek függvényében



8.5. ábra. A 8.4. táblázat adatainak grafikus reprezentációja

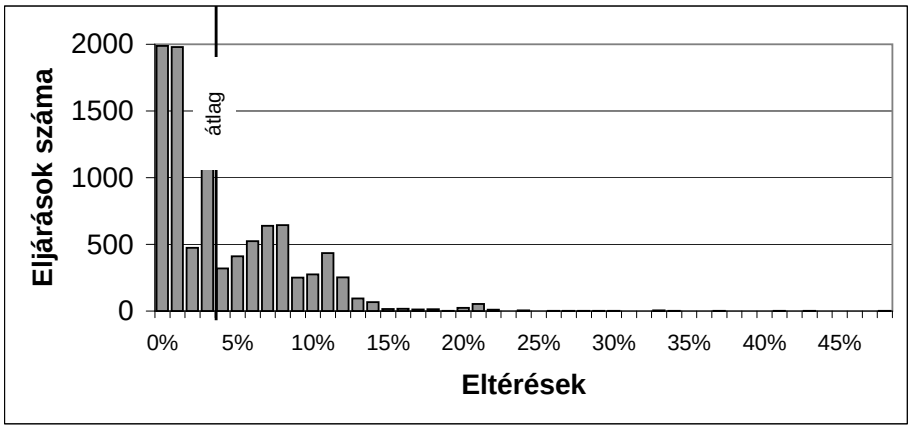
eredményét. Ez fontos megfigyelés, hisz ez azt jelenti, hogy egy potenciális változás hatáshalmazának nagyságát döntő módon az adat függőségek határozzák meg.

Az elvárásainknak megfelelően a fedettség értékei a SEB relációk által meghatározott hatáshalmazoknak a hátrahaladó szeletelés eredményeihez képest 100% volt minden esetben, és a pontosság értékek is igen magasnak bizonyultak. Eltekintve attól az esettől, amikor a pontosság érték 67%-os volt, az értékek 77,28% és 98,77% között mozogtak, ahogyan az leolvasható a 8.4. ábráról is. Így mondhatjuk, hogy a SEB reláció jól közelíti a hátrahaladó szeletelés eredményeit eljárás szinten, és így egy jó lehetőség lehet eljárás szintű hatásanalízis végrehajtásához.

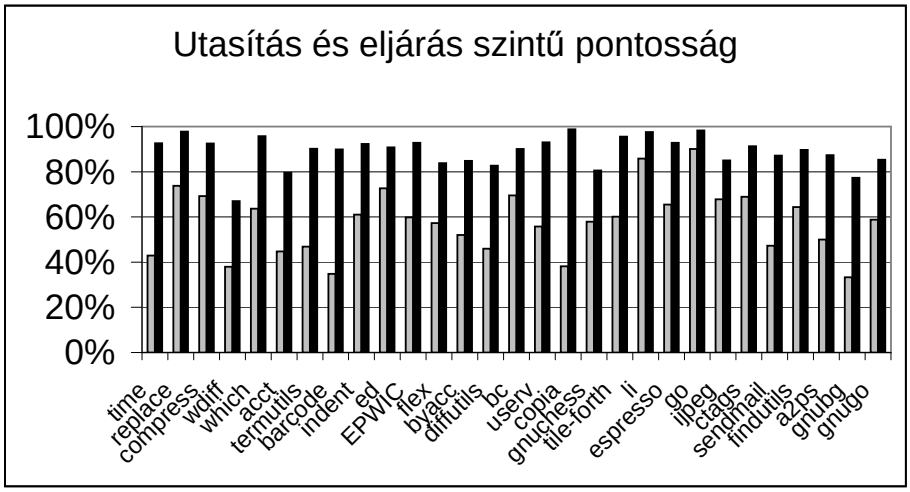
A statikus szeletelés legfőbb hátrányának sokszor azt mondják, hogy a kapott eredmény átlagosan felhasználhatatlan, mivel a programszelet majdnem az egész programot tartalmazza. A 8.4. táblázat adataiban bemutatjuk, hogy a vizsgált programjaink valamennyi eljárását nézve hogyan alakul az átlagos százalékos eloszlása azon eljárásoknak, amelyeket a vizsgált eljárás legalább egy kritériumából indított szelet érint. Ugyanebben

8.4. táblázat. Hátrahaladó szeletek és SEB halmazok átlagos méretei

Programok	Hátrahaladó szelet méretek	SEB méretek
a2ps	39,99%	46,78%
acct	56,40%	71,72%
barcode	14,72%	17,07%
bc	48,88%	54,22%
byacc	45,55%	54,74%
compress	44,97%	50,00%
copia	98,76%	99,99%
ctags	74,29%	80,55%
diffutils	27,80%	35,09%
ed	77,42%	83,70%
EPWIC	14,65%	16,63%
espresso	60,88%	65,51%
findutils	20,63%	22,77%
flex	63,92%	77,41%
gnubg	18,03%	23,20%
gnuchess	20,96%	24,84%
gnugo	21,84%	23,56%
go	95,69%	96,49%
ijpeg	34,71%	40,05%
indent	67,32%	71,13%
li	53,38%	54,50%
replace	72,34%	74,60%
sendmail	57,86%	66,39%
termutils	55,68%	62,88%
tile-forth	5,12%	5,61%
time	57,64%	63,89%
userv	47,14%	51,81%
wdiff	41,29%	64,75%
which	80,74%	84,06%



8.6. ábra. Eljárások SEB halmazainak és hátrahaladó szeleteinek különbsége az elemzett programoknál



8.7. ábra. SEB relációk utasítás- és eljárás szintű pontosságának összehasonlítása. A világos színű oszlopok a SEB relációk szeletekhez képest vett pontosságát adják meg utasítás szinten az egyes programok esetében, míg a sötét színű oszlopok ugyanezen értékeket jelenítik meg eljárás szinten.

a táblázatban a SEB relációk egy eljárásra jutó átlagos értékét is feltüntetjük. Könnyű látni, hogy van olyan eset, amikor az átlagos esetek is majdnem az egész programot magukba foglalják, de nem ez a tipikus, mint azt a bináris programoknál végzett méréseink is igazolják (lásd: 5.1. ábra).

Az eddigi adatok csak az átlagos értékeket közvetítették, és az átlagos szelet méretet és SEB halmaz méreteket hasonlították össze. Valójában a SEB relációkkal nemcsak átlagos esetben tudjuk a szeletelés eredményeit közelíteni, de minden eljárásra egyénileg is. A 8.6. ábra a hátrahaladó szeletek és SEB kapcsolatok eltéréseinek hisztogramját mutatja valamennyi vizsgált program valamennyi eljárására. Ahogy az leolvasható az ábráról, ezek az eltérések igen kicsik a legtöbb esetben. Habár van olyan eset, ahol a különbség a 48%-át is meghaladja az adott program méretének, ezeknek az eseteknek a száma eltörpül az összes eljárás száma mellett. Ráadásul megvizsgálva ezeket az eseteket, ahol a különbség nagyobb volt, mint 25%, azt kaptuk, hogy ezekért csupán három program volt a felelős (`wdiff`, `barcode`, `gnuchess`).

Az, hogy eljárás szinten a SEA (SEB) relációk meghatározása alkalmas arra, hogy a programszeleteket közelítsék, nem jelenti azt, hogy hasonló eredményt kapnánk, ha a relációnkat visszavezetnénk utasítások szintjére. Érdekességképpen összehasonlítottuk, hogy az utasítás szinten meghatározott SEB relációk, és programszelet méretek milyen kapcsolatban állnak egymással[2]. Függően az elemzési opcióknak, a CodeSurfer többkevesebb segéd csomópontot épít be a program gráf reprezentációjába utasítás szinten,

[2]Az A és B utasítások SEB relációban állnak egymással, ha van olyan út a program vezérlési gráfjában, ami tartalmazza az A és B utasítást is, ráadásul az A utasítás végrehajtása megelőzi a B utasítás végrehajtását.

8.5. táblázat. A különböző gráf reprezentációk méretei

valgrind	SDG	csomópontok száma	1 920 150
		élek száma	6 947 024
	ICCFG	csomópontok száma	154 509
		élek száma	179 626
gdb	SDG	csomópontok száma	10 086 409
		élek száma	48 876 108
	ICCFG	csomópontok száma	160 340
		élek száma	185 611
gcc	SDG	csomópontok száma	18 775 143
		élek száma	81 492 908
	ICCFG	csomópontok száma	467 185
		élek száma	584 972
mozilla	SDG	csomópontok száma	N/A
		élek száma	N/A
	ICCFG	csomópontok száma	1 587 499
		élek száma	1 723 611

így ezt az összehasonlítást csak azzal a megszorítással tehetjük meg, hogy ugyanazzal az elemzési beállításokkal határozzuk meg a szeleteléshez, illetve a SEB relációk meghatározásához szükséges gráfokat. Mivel az szeletelés igényli a speciálisabb beállításokat, így ennek megfelelően elemeztünk.

Az összehasonlítás során valamennyi eljárás valamennyi olyan csomópontjának meghatároztuk a hátrahaladó szeletét, illetve a vele SEB relációban álló csomópontok halmazát, amely csomópontok a vezérlési folyam gráf pontjai. Mivel a meghatározott programszeletek is tartalmazhatnak olyan csomópontokat, amik nem elemei a vezérlési folyam gráfnak – ilyenek például a formális és aktuális paramétereknek megfelelő csomópontok –, így az összehasonlítás előtt ezen csomópontokat is ki kell szűrni a kapott programszeletekből. Az összehasonlítás eredményét mutatja a 8.7. ábra, amelyen feltüntetjük az egyes programoknál az eljárás szinten számolt pontosság értékeket is. Nem meglepő, hogy utasítás szinten a programszeletek kevésbé helyettesíthetőek a SEA (SEB) relációkkal, hiszen a szeletelés egyébként is a program utasításai között teremt kapcsolatot, és ez önmagában sokkal precízebb, mint az a technika, amely csak közelíteni próbálja a szeletelés eredményét. Azonban eljárás szintre emelve a meghatározott kapcsolatokat az egyes utasítások által létrejött kapcsolatok összegződnek, így a közelítésük is sokkal pontosabb lesz.

Az a tény, hogy a SEB relációk meghatározásával hatékonyan tudjuk eljárás szinten követni a programszeletek alakulását még önmagában nem elegendő. A módszerünk és gráf reprezentációnk legfőbb előnye az, hogy sokkal egyszerűbb megvalósítani és előállítani, mint magát a szeletelést és az annak megfelelő gráf reprezentációt. Mind a szeletelés, mind a SEB relációk meghatározása tulajdonképpen egy-egy gráfbejárási algoritmus eredményeként adódnak, de nem mellékes szempont, hogy mekkora az adott gráf mérete, amin ezeket az algoritmusokat futtatjuk. A szeletelés, köszönhetően sokkal

8.6. táblázat. Különböző gráf reprezentációk előállítási ideje

System	Elemzési idő	SDG építési idő	ICCFG építési idő
valgrind	5 perc	16 perc	2 perc
gdb	8 perc	124 perc	4 perc
gcc	69 perc	571 perc	11 perc
mozilla	113 perc	N/A	54 perc

nagyobb gráfjának, sokkal több időt emészt fel. Habár a szeletelés költségesebb, mint a SEB relációk meghatározása, a szeletelés még mindig egy kényelmes megoldás lehet kisebb programok esetében a napjainkban használt számítógépek lehetőségeivel. Az olyan programok esetében azonban, amelyek eljárásainak száma a több ezret is meghaladja, és a sorok száma is eléri a milliót, a szeletelés általunk használt megvalósítása kivitelezhetetlen, míg a SEA (SEB) technika hatékonyan használható még mindig.

A méréseinkben használt program reprezentációk magasszinten ugyanolyan struktúrájúak, vagyis mindkettő valamennyi eljárást egy-egy csomóponttal reprezentálja. Az egy eljáráshoz kapcsolódó adatok tekintetében azonban már jelentősek a különbségek. A 8.5. táblázat összefoglalja ezeket az eltéréseket. Láthatjuk, hogy az ICCFG lényegesen kevesebb csomópontot és élt tartalmaz, ráadásul a `mozilla` esetében a CodeSurfer már arra sem képes a hatalmas méreteknek köszönhetően, hogy az SDG-t előállítsa. Így ezek az adatok hiányoznak is a táblázatból. A 8.6. táblázat a nagyobb rendszereket vizsgálja az egyes folyamatok végrehajtási ideje tükrében. Összehasonlításra kerülnek az egyes SDG-k és ICCFG gráfok előállításának idejei. A gráfméretek jelentős eltérései miatt a futásidők jelentős eltérései nem meglepőek annak ellenére sem, hogy a méréseket egy igen jó számítási kapacitású, AMD Opteron 2.2 GHZ-es, 4G memóriával rendelkező gépen végeztük.

8.5. Eredmények összegzése

Ebben a fejezetben olyan gyakorlati méréseket mutattunk be, amelyek a programszeletelés segítségével próbálják igazolni, hogy eljárás szinten a SEA, illetve SEB relációk meghatározása hatékony és pontos eszköz lehet a program eljárásai között létrejövő függőségek közelítésében. Gyakorlati méréseinkkel beláttuk, hogy a jóval egyszerűbben meghatározható kapcsolatok, mint amilyen a hívási relációk, nem alkalmasak valamennyi függőség megtalálására, ellentétben a SEA (SEB) relációkkal. A gyakorlati mérések megtervezése és kivitelezése a szerző önálló munkája.

9. fejezet

SEA és SEB relációk objektumorientált programokban

„Fontos, hogy mindent mérjünk, ami mérhető, és megpróbáljuk mérhetővé tenni, ami még nem az."

— Galileo Galilei

Ebben a fejezetben objektumorientált programok vizsgálatára fektetjük a hangsúlyt. A SEA relációk segítségével a vizsgált programok osztályai között próbálunk olyan rejtett függőségeket, kapcsolatokat feltárni, amely kapcsolatokat a hagyományos csatolási metrikák nem fedeznek fel, illetve a program szintaktikai elemzésével nem jönnek elő. Vizsgáljuk a rejtett függőségek és az expliciten megjelenő függőségek kapcsolatát is egyéb objektumorientált metrikával együtt.

9.1. Rejtett függőségek és meghatározásuk SEA relációk segítségével

A célunk a rejtett függőségek vizsgálatakor az volt, hogy objektumorientált programok osztályai közötti összes lehetséges kapcsolatot összegyűjtsük. Ezt a szándékunkat az tette indokolttá, hogy számos alkalmazás, mint amilyen például a változás propagáció, a regressziós tesztelés szintén osztály szintű elemzését kívánják meg az objektumorientált rendszernek, ráadásul úgy, hogy az összes valódi függőség ismert legyen előttük. Számos lehetőség van arra, hogy az expliciten létrejövő kapcsolatokat kinyerjük egy adott programból, de sokszor ezen kapcsolatok feltérképezése nem elegendő.

A 9.1. ábrán bemutatunk egy szemléletes példát a rejtett függőségekre. Ebben a példában az `A` osztály tartalmaz egy `int A::get()` metódust, ami egy felhasználó ál-

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

9.1. ábra. Példa rejtett függőségre

tal adott színt kódol úgy, hogy a „0" érték jelöli a pirosat, az „1" a sárgát, és így tovább. A `B` osztály tartalmaz egy `void B::paint(int)` metódust, ami a kapott kódnak megfelelően a képernyőt kifesti a neki megfelelő színre. A `C` osztály pedig tartalmaz egy adat folyamot az `A` osztály és a `B` osztály egy-egy példánya között. Amennyiben a programozó megváltoztatja az egyes színeknek a kódolását az `A` osztályban, úgy értelemszerűen meg kell változtassa a `B` osztály működését is, és fordítva. Annak ellenére, hogy direkt módon az `A` és `B` osztályok nem hivatkoznak egymásra, köztük létezik függőség, amire mi rejtett függésként tekintünk, hiszen expliciten nem észrevehető ez.

A már bevezetett SEA és SEB relációk alkalmasak lehetnek a rejtett függőségek közelítésére. A SEA relációk meghatározásával észrevehetjük, hogy az `a.get` hívást követi a `b.paint` meghívása, és így a `B` osztály függhet az `A` osztálytól. Mivel egyébként a két osztály látszólag nem kapcsolódik egymáshoz, a köztük lévő SEA reláció miatt azt mondhatjuk, hogy köztük rejtett függőség létezhet.

Ahhoz tehát, hogy osztályok közötti összes függést, illetve kapcsolatot megadjuk, az eredetileg eljárások között értelmezett SEA és SEB relációkat terjesztjük ki osztály szintre. Ennek megfelelően az elemzéseinkben egy `B` osztályra akkor mondjuk, hogy függ az `A` osztálytól, ha van olyan lehetséges lefutása a programnak, ahol a `B` osztályt érintő bármely esemény az `A` osztály valamely eseménye után következik[1].

Csupán az eljárások közötti SEA és SEB relációk osztály szintre való kiterjesztése nyilván nem határozza meg az összes lehetséges függőséget, hiszen ebben a formában csak az eljáráshívások által megvalósult kapcsolatokat tudjuk összegyűjteni. Azt, amikor egy osztály egyik adattagját közvetlen érjük el egy másik osztályban, nem kezeli le ez a technika, illetve a globális változók és osztályok közötti kapcsolatokat sem. A globális változók és a tagváltozók közvetlen elérésével létrejött kapcsolatok láthatatlanok maradnak az algoritmus számára, hiszen az ezeknek megfelelő csomópontok hiányoznak

[1] Egy osztályhoz kapcsolható eseménynek tekintjük bármely metódusának végrehajtását, de ezen kívül természetesen bármely adattagjának használata, definiálása is az.

Eredeti kód

```
int global=0;

class A {
  public: int i;
};

class B {
  A a;
  public: int foo() {
    return global+a.i;
  }
};
```

Módosított kód

```
struct globalStruct {
  static int global;
  static int& get_global() { return global; }
};
int globalStruct::global = 0;

class A {
  public: int i;
  int& get_i() { return i; };
};

class B {
  A a;
  public: int foo() {
    return globalStruct::get_global()
           + a.get_i();
  }
};
```

9.2. ábra. Globális változók és tagváltozók kezelése

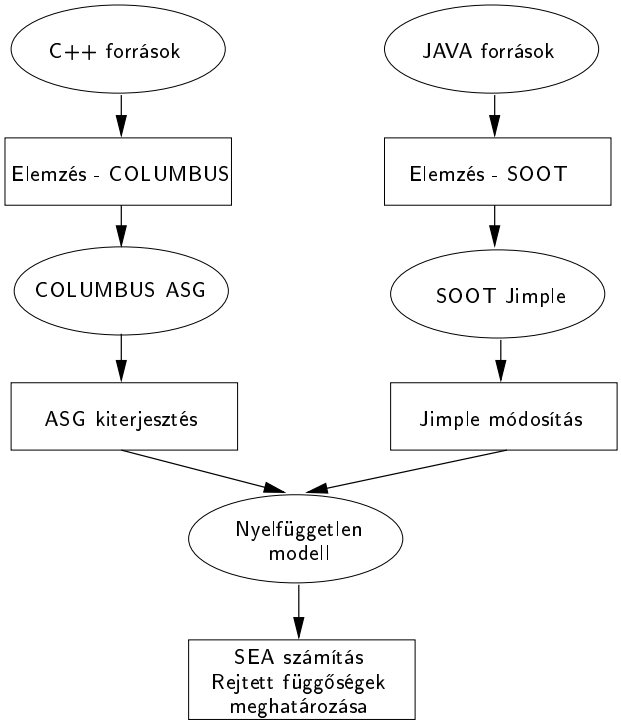
az ICCFG gráfból.

Ezt a problémát kiküszöbölhetjük azzal az egyszerű trükkel, hogy még az ICCFG gráf építése előtt minden globális változót beágyazunk egy olyan általunk létrehozott osztályba, amelynek megfelelő *set* és *get* metódusaival definiálni, illetve használni tudjuk az adott változót. Ugyanígy minden osztály adattaghoz létrehozunk egy-egy *set* és *get* metódust, amennyiben nem rendelkezik ilyenekkel. Minden alkalommal, amikor egy `X` változót (legyen az globális változó, vagy tagváltozó) olvasnánk, lecseréljük az `X` ezen előfordulását egy `get_X()` hívásra, míg minden helyen, ahol definiálnánk egy adott e értékkel, azt a `set_X(e)` hívással helyettesítjük. A 9.2. ábra konkrét esetben mutatja be az elemzett programon végzett transzformációkat.

Az elemzendő program előbb bemutatott átalakítása után felépítjük az ICCFG gráfot, majd meghatározzuk a SEA, illetve SEB relációkat. Ezek után már mondhatjuk, hogy a `B` osztály akkor és csak akkor függ az `A` osztálytól, hogyha van olyan f metódusa az `A` osztálynak, amivel a `B` osztály egy g metódusa SEB relációban áll, illetve ahol a g metódus SEA relációban áll az f metódussal.

9.2. Gyakorlati mérések megvalósítása

Azt már nem kell megmutatnunk, hogy a SEA és a SEB relációkkal hatékonyan tudjuk közelíteni a programban rejlő valódi függőségeket, hisz ezt már megtettük az előző, 8.4. fejezetben. Célunk, hogy megmutassuk, hogy objektumorientált nyelveknél használt egyéb megvalósításoknál sokkal megbízhatóbb eredményt tudunk előállítani az ál-



9.3. ábra. Mérési keretrendszer

talunk bevezetett technikával olyan alkalmazások esetén, mint a hatásanalízis, változás propagáció, illetve regressziós tesztelés. Ahhoz, hogy megmutassuk, hogy a SEA relációk meghatározása általánosan alkalmas arra, hogy az objektumorientált programokban rejlő rejtett függőségeket hatékonyan, és pontosan közelítse, úgy implementáltunk a SEA relációkat és a SEA alapú rejtett függőségeket meghatározó eszközünket, hogy az az elemzendő program egy nyelv független reprezentációját használja fel a számítások alapjául. Ez azt jelenti, hogy az alap algoritmus tetszőleges objektumorientált program rejtett függőségeit képes meghatározni, elegendő csupán a frontendet lecserélni, ami az adott nyelvnek megfelelő nyelvtan alapján elemzi a kódot. A relációkat először metódus szinten határozzuk meg, és ezeket a kapcsolatokat terjesztjük ki osztály szintre.

A gyakorlati méréseket C++ és Java nyelvű programokon végeztük. A 9.3. ábrán összefoglaljuk a gyakorlati mérések főbb lépéseit. A C++ programokat a Columbus eszközzel elemeztük [25], míg a Java programokat a Soot Java elemzővel [55]. A C++ programok elemzése után egy absztrakt szintaxis fa tartalmazza a program reprezentációját, míg Java esetén az elemzéssel egy egyszerűsített, háromcímes program reprezentációt, a Jimple-t kapjuk. Ezeken a reprezentációkon kell megvalósítanunk azokat a program transzformációkat, amit az előzőekben bemutattunk, azaz C++ esetén a globális változókat, illetve a tagváltozók direkt elérését kell lecserélnünk, míg Java esetében nyilvánvalóan csak ez utóbbival kell foglalkozzunk. A módosított reprezentációkból már egyszerűen elő tudjuk állítani azt a nyelv független modellt, ami tulajdonképpen nem más, mint az ICCFG gráfunk, azzal a kiegészítéssel, hogy minden metódust reprezentáló

csomópont címkézve van azzal az információval, hogy melyik osztályba tartozik.

9.3. Függőségek szűrése

Minden összetettebb szoftver rendszer sok-sok osztályból áll, amelyek jelentős része könyvtári osztály. Mivel célunk, hogy azokat a függőségeket határozzuk meg, amik a programozó számára lényegesek lehetnek a szoftverfejlesztés során, elképzelhető, hogy a felesleges függőségeket ki kell szűrjük. Ilyenek lehetnek a könyvtári osztályokhoz kötődő kapcsolatok. Arra, hogy miképpen szűrünk, több lehetőségünk is van.

Valamennyi osztály vizsgálata. Ennek az eredménye függőségeknek egy olyan hatalmas halmaza, amely valamennyi függőséget tartalmazza, még azokat is, amik a könyvtári osztályok között valósulnak meg. Mivel átlagosan ezekkel az osztályokkal semmit nem tudunk tenni a fejlesztés során, és ezek átlagosan ugyanazok tetszőleges programok esetében, így nyilvánvaló, hogy ez a megközelítés finomításra szorul.

Vágás a számítások előtt. Ebben az esetben a függőségeket csak azokra az osztályokra számítjuk ki, amelyek az elemzett rendszer részei, a többi osztályra nem. Ez azt jelenti, hogy a SEA relációkat is csak az előbbi osztályokra határozzuk meg. Azonban ha a könyvtári osztályokat teljesen mellőzzük a számításainkban, elképzelhető olyan eset, hogy lényeges rejtett függőséget nem találunk meg a rendszer osztályai között. Például ha egy könyvtári metódus callback módon hívja meg valamely rendszerbeli osztályt, vagy egy könyvtári osztályból származtatott rendszerbeli osztály valamely metódusa a könyvtári osztály valamely másik metódusából hívódik meg polimorfikusan, akkor elképzelhető, hogy lesz olyan `A` és `B` osztálya a rendszerünknek, amelyek közötti rejtett függőséget nem ismerjük fel.

Vágás a számítások után. Ez az út az előző két eset kombinálása. Vagyis a függőségeket a teljes rendszerre számítjuk, de miután a függőségeket meghatároztuk, az összes olyan kapcsolatot elvetjük, amely könyvtári osztályokhoz kötődik. Így biztosan nem hagyunk el olyan függőséget, amely az adott rendszer osztályai között jelentkezik. A méréseinkben mi ezt a harmadik változatot használtuk.

9.4. Gyakorlati mérések kiértékelése

Amikor egy alkalmazásban az egymástól függő osztályokat keressük, azt kell eldöntenünk, hogy a függőségi relációra szimmetrikus relációként tekintünk, vagy sem. Más szóval azt kell eldönteni, hogy a függőségi relációk, amelyeket meghatározunk irányított kapcsolatot írnak le, vagy sem. Különböző alkalmazásokat nézve elképzelhető, hogy különbséget

9.1. táblázat. Teszt programok

Program	Nyelv	Osztályok/struktúrák/uniók száma
unrar	C++	72
mysqlcc	C++	135
licq	C++	172
stellarium	C++	203
AbsHiddenDep	C++	6
AbsHiddenDep*	C++	406
JSubtitles	JAVA	14
RayTracer	JAVA	12
java2html	JAVA	50
dynjava	JAVA	301

kell tegyünk ezen a téren. Például változás propagáció során az osztályok között felismert függőségekre szimmetrikus relációként kell tekintenünk, hiszen ha egy változtatást teszünk a program egy adott osztályában, akkor lehet, hogy a változtatásnak megfelelően mind azokat a programrészeket módosítanunk kell, amelyek végrehajtása befolyásolja az adott osztály viselkedését, mind azokat, amit az adott osztály befolyásol. Hatásanalízis során azonban lehet csak az az irány érdekes a felhasználás szempontjából, hogy adott programponttól, adott osztálytól mely másik osztályok függenek. Amikor a gyakorlati mérésünkben a *CBO (Coupling Between Object)* metrika értékkel hasonlítjuk össze az általunk számított függőségi relációkat, akkor is nyilvánvalóan nem tekinthetünk a saját relációnkra sem úgy, mintha szimmetrikus reláció lenne, hiszen a CBO sem az, és így félrevezető eredményt kapnánk.

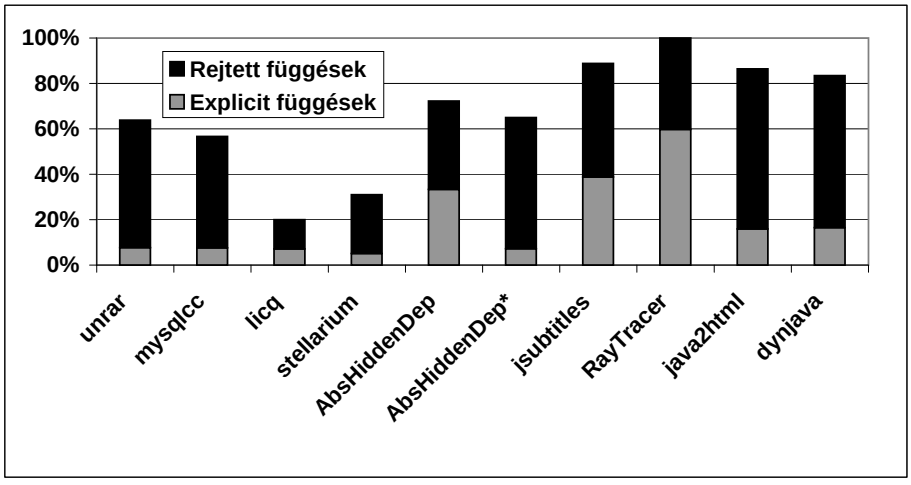
A gyakorlati méréseinket közepes méretű C++ és Java programokon végeztük, amelyek jellemzőit a 9.1. táblázat foglalja össze. A C++ programok esetében köszönhetően a Columbus elemző eszköznek [25] az egyes osztályokhoz tartozó CBO metrikák meghatározása is lehetséges volt számunkra, így az összehasonlítás végett ezen értékeket is meghatároztuk ezekre a programokra. Az elemzett programok nyíltforrású projektekből álltak össze, kivéve az AbsHiddenDep programot, ami a saját programunk, amely megvalósítja a SEA technikát. Ezt a programot ráadásul két különböző módon vizsgáltuk. Egyik esetben csupán önmagában, másik esetben viszont a Columbus keretrendszer összes kapcsolódó könyvtárával együtt. Ez utóbbi esetet egy *-gal különböztettük meg az ábrákon és a táblázatokban.

A méréseink első részében meghatároztuk a kiválasztott programoknál az explicit osztály kapcsolatok és a rejtett osztály kapcsolatok számát, valamint az egy osztályra jutó átlagos kapcsolatok számát. Ennek a mérésnek az eredményét foglalja össze a 9.2. táblázat, míg a 9.4. ábra grafikusan ábrázolja ugyanezeket az adatokat a program teljes méretéhez viszonyítva. Ahogy az észrevehető általánosságban sokkal több a rejtett függőség, mint azt talán vártuk volna.

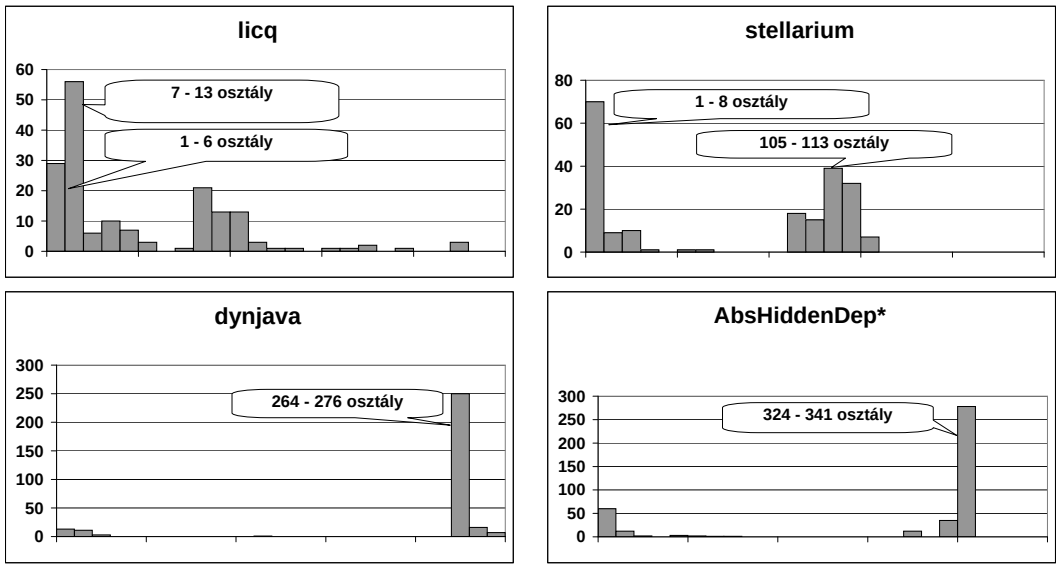
Az összes és átlagos számok mellett sokkal átfogóbb képet ad, ha azt is vizsgáljuk,

9.2. táblázat. Függőségek száma a teszt programokban. A zárójelezett számok az egy osztályra jutó átlagos függőségek számát mutatják, míg a zárójelen kívül értékek a programban megtalálható összes függőségeknek a száma.

Program	Expliciten függő osztálypárok száma	Rejtett függésben álló osztálypárok száma
unrar	396 (5.5)	2912 (40.44)
mysqlcc	1381 (10.23)	8952 (66.31)
licq	2082 (12.1)	3822 (22.22)
stellarium	2083 (10.26)	10700 (52.7)
AbsHiddenDep	12 (2)	14 (2.33)
AbsHiddenDep*	11872 (29.24)	95158 (234.38)
JSubtitles	76 (5.43)	98 (7)
RayTracer	86 (7.17)	58 (4.8)
java2html	398 (7.96)	1762 (35.24)
dynjava	14923 (49.58)	60672 (201.56)



9.4. ábra. Átlagos függőségek osztályonként az összes osztály számához viszonyítva

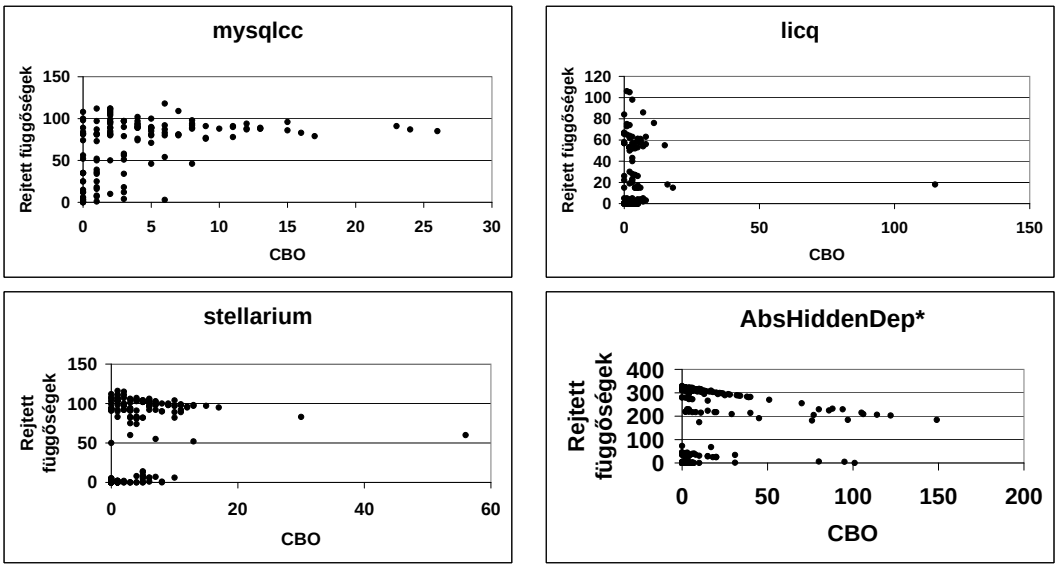


9.5. ábra. Függőségek számának eloszlása. A hisztogramok vízszintes tengelye az egyes osztályokhoz tartozó függőségek számát követi növekvő sorrendben, míg a függőleges tengely azt adja meg, hogy adott számú függőséggel hány osztály rendelkezik az adott rendszerben.

hogy milyen az eloszlása ezeknek az értékeknek az egyes osztályok tekintetében. A 9.5. ábra ezt az eloszlást az egy osztályra jutó függőségek hisztogramján mutatja a nagyobb programok esetén. Egy érdekes megfigyelés, hogy a bemutatott négy eset közül háromban kiugró részeket találunk nagyobb osztályszámok esetén is. Ha alaposabban megvizsgáljuk ezeket az eseteket, akkor azt kapjuk, hogy ez a szituáció majdnem ugyanaz, amivel más szerzők már foglalkoztak programszeletek méreteinek összehasonlításakor [17; 57]. Az ilyen kiugró részek úgynevezett *függőségi klasztereket* határozhatnak meg, ahol nemcsak az igaz, hogy a függőségek száma megegyezik, de az is nagyon valószínű, hogy az egyes osztályok ilyenkor ugyanazon másik osztályoktól állnak függésben. A szoftverfejlesztés, illetve karbantartás során az ilyen klaszterek jelenléte igen nagy probléma, hiszen bármilyen változtatás a klaszter egy elemén kihatással lehet a klaszter további pontjaira, amelyek működését ezután ellenőrizni kell. Minél nagyobbak ezek a klaszterek, annál költségesebb egy-egy klasztert érintő módosítás kivitelezése, tesztelése.

Briand és társai egy olyan modellt adtak a hatásanalízishez, amelyben csatolási mértékek alapján határozták meg egy adott osztály hatáshalmazát. Vizsgálataikban arra mutattak rá, hogy csupán a direkt függőségeket mérő metrikák, mint amilyen az objektumok közötti csatolás, vagyis a CBO, nem elegendőek arra, hogy valamennyi változást meghatározzanak a segítségükkel. Annak érdekében, hogy ezt a feltételezésüket bizonyítsuk, összevetettük az egyes osztályokhoz tartozó CBO értékeket az általunk meghatározott rejtett kapcsolatok számával[2].

[2]Megjegyezzük, hogy az explicit függőségek száma erősen korrelál a CBO értékkel, de ezek azért nem ugyanazok, köszönhetően a meghatározásuk módjának. Az összehasonlításunkban azért választottuk a



9.6. ábra. Kapcsolat a CBO metrika értékek és a rejtett függőségek száma között. Az ábrán minden pont az elemzett rendszer egy osztályának felel meg. Míg a vízszintes tengely mentén a rendszer minden osztályát a rejtett függőségeinek függvényében, addig a függőleges tengelyen a CBO értékének függvényében ábrázoljuk.

Mivel a CBO számításakor a létrejött kapcsolatokat irányítottaknak tekintjük, így az összehasonlítás során mi is csak irányított kapcsolatokat határozunk meg a SEA relációkra építve. 9.6. ábra mutatja a legtipikusabb eseteket, amivel a vizsgált programoknál találkoztunk. Nem elfeledve, hogy a SEA relációk behozhatnak hamis függőségeket is, kijelenthetjük, hogy a hipotézis arról, hogy a CBO önmagában nem elegendő a legtöbb esetben, igaz. Találhatunk olyan osztályokat a vizsgált rendszerekben, amelyeknek mind a CBO értéke, mint a rejtett függőségeinek a száma alacsony, de a legérdekesebb területek azok, ahol az alacsony CBO értékekhez sok rejtett függőség tartozik. Ez pedig azt sugallja, hogy annak ellenére, hogy a rejtett függőségek között hamis függőségek is bekerülhetnek, a CBO csak alulról becsli a valódi függőségeket, és így a hatásanalízis, hiba előrejelzés területén használhatatlan eszköz önmagában.

9.5. Eredmények összegzése

Ebben a fejezetben megadtuk azokat a szükséges program transzformációkat, amelyek alkalmassá teszik a SEA (SEB) relációkat arra, hogy objektumorientált programok között létrejött függőségeket közelítsék. Mivel objektumorientált programok esetében az expliciten meghatározható objektumorientált metrikák használatosak arra, hogy az egyes osztályok között létrejövő függőségeket jelezzék, ennek a fejezetnek a fő célja az volt, hogy gyakorlati mérésekkel mutassa meg, hogy valójában milyen pontosak ezek a módsze-

CBO-t az explicit függőségek helyett, mert a gyakorlatban ez a leginkább elfogadott csatolási mérték.

rek. Mivel a 8. fejezet gyakorlati méréseivel már beláttuk, hogy a SEA (SEB) relációk alkalmasak a programban lévő függőségek biztonságos közelítésére, így ebben a fejezetben csak azt vizsgáltuk, hogy a SEA (SEB) relációk által meghatározható osztályszintű kapcsolatok milyen viszonyban állnak az egyéb objektumorientált metrikákkal. A gyakorlati mérések igazolták, hogy az expliciten megjelenő függőségek száma jóval kevesebb, mint a rejtett függőségek száma. Ez azt jelenti, hogy önmagukban ezek a metrikákon alapuló módszerek nem adnak megbízható eredményt olyan alkalmazásokban, amelyek működése az osztályok közötti függőségek meghatározásán alapul. A mérések megtervezése a kapcsolódó publikáció szerzőivel közös eredmény, míg a C/C++ programokhoz kapcsolódó mérések kivitelezése, valamint a SEA (SEB) relációk gyakorlati megadása a szerző önálló munkája.

10. fejezet

Eredmények összegzése

„A végkifejlet igazolja a cselekményt."

— Ovidius

Az értekezés első részében bemutattuk a bináris programok statikus szeletelésének problémáit, valamint adtunk egy konzervatív, függőségi gráfok bejárásán alapuló szeletelési megoldást. Annak ellenére, hogy a bináris programok szeletelése szélesebb körben felhasználható, mint a magasszintű nyelvek szeletelése, ezzel a témával nemigen foglalkozik a kapcsolódó irodalom. A megadott algoritmust statikus és dinamikus adatokkal tovább pontosítottuk azzal, hogy a függőségi gráfból elhagytunk olyan éleket, amelyek mögött valódi függőség nem lehetett.

Valamennyi megoldásunkat implementáltuk, majd kiértékeltük ARM architektúrára fordított programokon. A konzervatív megoldást választva átlagosan 52%-os szelet méretet értünk el, amely azt igazolja, hogy bináris programok statikus szeletelése hatékony eszköz lehet számos rá épülő alkalmazásban.

Különböző módszerekkel megpróbáltuk elérni, hogy az eredményül kapott szelet méretek még kisebbek legyenek. A függőségi gráfok statikus pontosításával 1% - 4 %-os szelet méret csökkenést sikerült elérni. A relatív kis mértékű javulás a memória-elérések konzervatív kezelésének, és a nagyszámú indirekt függvényhívásoknak köszönhető.

Dinamikus adatok segítségével bemutattuk, hogy a szelet méretét drámaian csökkenthetjük, amennyiben az elemzett program hívási gráfját pontosítjuk. Hátránya a dinamikus adatokkal támogatott elemzésnek, hogy az már nem biztonságos, azaz elképzelhető, hogy általa valódi függőségeket nem veszünk figyelembe. Kellő számú dinamikus adat felhasználásával, elegendő teszt esettel olyan kódlefedettséget érhetünk el, amely a valódi hívási gráfot jól közelíti. Néhány alkalmazásban, például hibakereséskor, nem követelmény az, hogy a kapott szelet biztonságos legyen, hiszen elég akkor foglalkozni a pontosabb gráf reprezentációval, ha a kevésbé pontos gráf használata nem járt eredménnyel.

Az értekezés második részben bevezettük a Statikus Execute After és a Statikus Execute Before relációkat, amelyeket magasszintű nyelvek elemzésével vizsgáltunk. Bevezettünk egy olyan programreprezentációt, amely alkalmas arra, hogy segítségével ezen relációkat megadjuk, ráadásul több módszert is mutattunk ezen relációk meghatározására. Habár ezen relációk meghatározása számos olyan kapcsolatot adhat, amelyek valódi függéssel nem bírnak, használatuk mégis előnyös lehet olyan függőségek feltérképezésében, ahol bár szintaktikus kapcsolat nincs két program komponens között, de valamilyen szemantikus viselkedés miatt mégis létezik függőség közöttük.

Gyakorlati mérésekkel beláttuk, hogy annak ellenére, hogy a Statikus Execute After és a Statikus Execute Before relációk tartalmazhatnak nem valódi függőségeket, mégis nagy pontossággal közelítik eljárás, vagy magasabb szinten a statikus programszeletelés eredményét. Mivel ezen relációk meghatározása kevésbé költséges, mint a szeletelés megvalósítása, így hasznos eszköz lehet nagyon sok alkalmazásban, ahol a statikus szeletelést helyettesíthetik. Gyakorlati méréseink azt is igazolták, hogy a SEA (SEB) relációk és az objektumorientált programok osztályainál létrejött direkt és indirekt csatolások között szignifikáns korreláció figyelhető meg. Azzal, hogy ezen relációkkal olyan rejtett függőségek is felfedezhetőek, amelyeket más, egyszerűbb módszer nem talál meg, alkalmassá teszi ezen relációkat, hogy olyan alkalmazásokat tegyenek megbízhatóbbakká, mint amilyenek többek között a hatásanalízis, változás propagáció, tesztelés.

A. Függelék

Futtatható programszeletek a gyakorlatban

„A világot egy hiba lendítette mozgásba - sose félj hibázni."

— Paulo Coelho

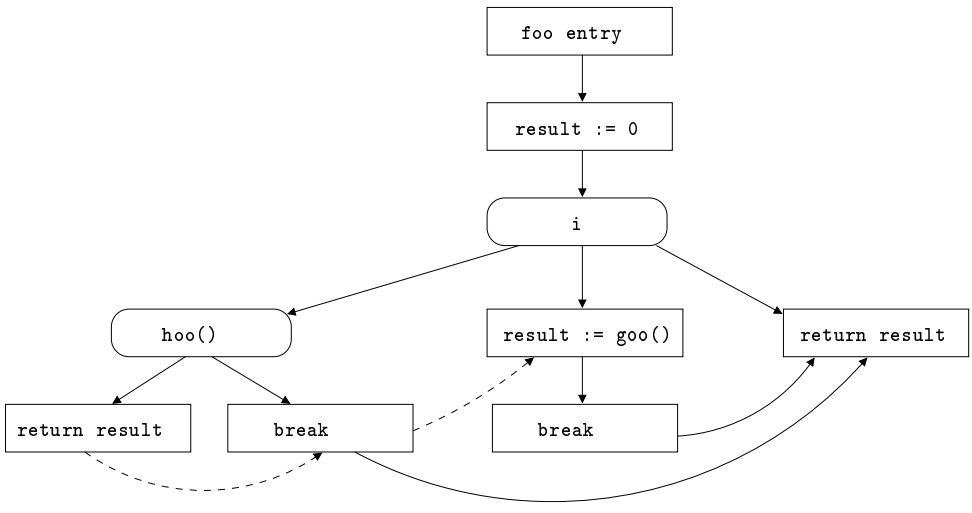
A függeléknek ebben a részében ismertetjük a CodeSurfer programszeletelő eszköz [29] azon hibáját, és a hiba okait, valamint hatásait, amely révén a 8.4. fejezetben bemutatott méréseinket csak a SEB relációk és hátrahaladó szeletek között tudtuk elvárásainknak megfelelően elvégezni. Hatásanalízis szempontjából bár az előrehaladó szeletek és SEA relációk összehasonlítása lett volna indokoltabb összehasonlítás, nem ezt az utat választottuk a gyakorlati eredmények összevetésénél, hiszen bizonyos esetekben az előrehaladó szeletelés pontatlanságából adódóan olyan eltérések jelentkeztek a előrehaladó szelet és SEA halmazok között, amelyek torzították az eredményeket. Valójában egy megfelelően működő statikus szeletelőnél igaznak kell lennie annak, hogy az előrehaladó szeletelés a hátrahaladó szeletelés duális párja. Az általunk elvégzett mérések mivel valamennyi csomópont szeleteit vizsgálták, így összességében ha a dualitás teljesült volna, az átlagos eredmények minden esetben megegyeztek volna az előrehaladó és hátrahaladó esetekben.

A.1. Futtatható programszeletek számítása

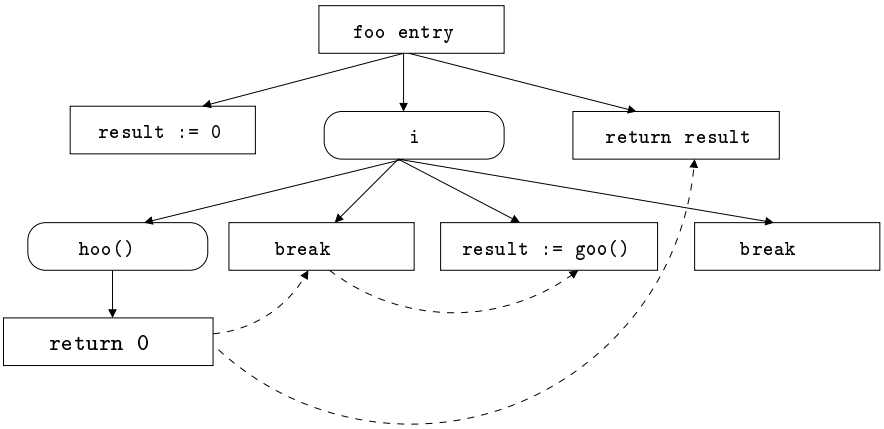
Ebben a fejezetben azt vizsgáljuk, hogy a szeletek között lévő duális tulajdonság miért sérülhetett. A CodeSurfer szeletelési algoritmusa Ball és Horwitz algoritmusára épül [7]. Annak érdekében, hogy futtatható szeletet tudjanak előállítani, kiegészítik a program

```
1   int foo(int i){
2     int result := 0;
3     switch(i){
4       case 1:
5         if (hoo())
6           return 0;
7         break;
8       case 2:
9           result = goo();
10          break;
11      }
12    return result;
13  }
```

A.1. ábra. Példa Bent és társai nyomán



A.2. ábra. Az A.1. ábra kódjához tartozó vezérlési folyam gráf. A folytonos élek a vezérlési folyam éleket, a szaggatott élek a kiterjesztett vezérlési éleket jelölik.



A.3. ábra. Az A.1. ábra kódjához tartozó programfüggőségi gráf. A folytonos élek a vezérlés függőségeket, a szaggatott élek hamis vezérlés függőségeket jelölik.

eredeti vezérlési folyam gráfját további élekkel, amelyek ugró vagy abortáló utasításokkor (`break, continue, goto, return ...`) rámutatnak azokra az utasításokra, amelyekre normálisan kerülne a vezérlés, ha nem egy ugró utasítás lenne az adott ponton. Így az ilyen utasításoktól lényegében vezérlés függésben állnak azok, amelyekre az ugrás miatt nem kerül a vezérlés. Ez az eljárás jó, ha olyan futtatható programszeletet szeretnénk előállítani, amelyet utasítások törlésével kapunk az eredeti programból, és az előálló program szemantikailag megegyezik az eredeti programmal az adott programpont tekintetében.

Bent és társai különböző szempontok alapján összehasonlították a CodeSurfer egy korábbi verzióját az általuk megvalósított Sprite szeletelővel [8]. Vizsgálták, hogy egy-egy függőség pontosabb megadása milyen mértékben hat a programszeletek méretére, illetve példát adtak arra, hogy bár az esetek többségében a CodeSurfer sokkal pontosabb, mint a saját maguk által kifejlesztett Sprite szeletelő eszköz, de köszönhetően annak, hogy a CodeSurfer arra törekszik, hogy futtatható programszeletet állítson elő, sokszor olyan utasítások is belekerülnek nála a szeletbe, amik valójában nem határozzák meg az adott programpont viselkedesét, és amelyek akár el is hagyhatóak lennének a szeletekből. Az A.1. ábra példájának 9. sorából indított hátrafelé haladó szelet eredetileg tartalmazza az 5. és 6. sorok utasításait is, annak ellenére, hogy ezeknek semmilyen hatása nincs a kritériumra, és elhagyásuk mellett is a keletkezett szelet szemantikus viselkedése a kritériumként megadott pontban megegyezik az eredeti program viselkedésével. Hasonlóan az 5. sor feltételéből kiinduló előrehaladó szelet feleslegesen tartalmazza a 9. sor utasítását.

A probléma visszavezethető [7] algoritmusára. Vegyük a `foo` függvény vezérlési folyam gráfját, és egészítsük ki azt az algoritmusban megadott vezérlési élekkel! Építsük fel a programfüggőségi gráfot, figyelembe véve a kiegészített vezérlési éleket is, amelyek által *hamis vezérlés függőségek* is bekerülnek! Az A.3. és az A.2. ábrák mutatják

A.1. táblázat. Előrehaladó szeletek szempontjából problémás tesztesetek

Programok	Különbségek az előrehaladó és hátrahaladó szeletek között	Előrehaladó szeletbe helytelenül bekerült elemek aránya
termutils	0,15%	0,276%
ed	0,39%	0,482%
diffutils	0,50%	1,760%
espresso	5,35%	8,027%
sendmail	0,02%	0,028%
a2ps	4,25%	9,423%
gnubg	0,43%	2,240%
gnugo	0,69%	2,764%

az A.1. ábra kódjának ezen gráfjait[1].

Ball és társai ezt a hamis vezérlés függőséget, mint egy nem tranzitív relációt definiálják, a szelet meghatározását viszont Horwitzék szeletelésének módjára adják meg [32], vagyis a szeletelés itt sem más, mint egy elérési probléma ezen a programfüggőségi gráfon. Mivel a hamis vezérlés függőség nem tranzitív reláció, így olyan utakat nem kellene figyelembe venni, ahol az útvonal két szomszédos elpárja ilyen él. Ezek kiküszöbölésével úgy csökkenthetjük a bejárás által kapott eredményt, hogy közben a kapott szelet maradéktalanul teljesíti azt a követelményt, hogy a szelet által behozott utasítások szemantikailag ugyanazt a programot adják a kritérium tekintetében, mint az eredeti program.

A CodeSurfer jelenlegi - 2.1p1 verziója - bár hátrahaladó szeletek esetében figyel erre, előrehaladó esetekben nem feltétlen. A GrammaTech ígérete alapján ezt a következő release változatban már korrigálják.

A.2. A probléma hatása a gyakorlati eredményekre

A fent bemutatott probléma hatása, hogy az előrehaladó és a hátrahaladó szeletek dualitása megszűnik bizonyos esetekben. Azaz elképzelhető, hogy van olyan A programpont, amely előrehaladó szeletében benne van a B programpont, de a B pont hátrahaladó szeletében nincs benne az A pont. Ráadásul az előrehaladó szeletek és SEA viszonylatában is gondot okoz ez a különbség, hiszen így voltak olyan eljárások, amelyeknek olyan eljárás is belekerült az előrehaladó szelet halmazába, ami a SEA halmazok között nem jelent meg[2].

Az A.1. táblázat mutatja azokat a teszteket, amelyeknél a probléma kézzel fogható volt. Mivel valamennyi lehetséges csomópontból indítottunk szeleteléseket, így azt vár-

[1] Az adatéleket nem ábrázoljuk, ezeknek a problémát tekintve nincsen most jelentőségük.

[2] Ez persze így jogos eltérés volt, hiszen egy eljárás SEA halmazába nem kell hogy bekerüljenek azok az eljárások, amelyek nem érhetőek el az adott eljárásból indulva vezérlési folyam éleken keresztül.

tuk, hogy a szeletek összértékei a két irányban meg fognak egyezni minden esetben. A táblázat második oszlopa azt gyűjti össze, hogy ezekben az esetekben ez hány százalékkal növelte meg az előrehaladó szeletek átlagos, egy eljárásra jutó méretét az egyes programoknál. A harmadik oszlop pedig azt mutatja, hogy a SEA által igazoltan hamis kapcsolatok hány százalékát teszik ki az előrehaladó szeletek által meghatározott kapcsolatoknak. Összesen 8 program esetében jött elő a probléma, és olyan is akadt a `li` program, ahol egyetlen rosszul beazonosított kapcsolat volt, néhány esetben azonban a feleslegesen megjelenő kapcsolatok száma jelentős mértékű.

B. Függelék

Magyar nyelvű összefoglaló

„A valósághoz mérten minden tudományunk primitív és gyerekes dolog – és mégis ez legnagyobb értékünk."

— Albert Einstein

Bevezetés

Az értekezés meghatározó témája a programszeletelés, illetve közvetve a program komponensei közötti függőségek vizsgálata. Míg az értekezés első részében függőségi gráfok segítségével határozzuk meg a bináris program szeleteit, addig a második felében egy általunk bevezetett függőségi módszert tesztelünk a programszeletelés eredménye által. Azaz a cél és eszköz szerepet cserél a két részben. Az első részben a célunk függőségek alapján meghatározni bináris programok szeleteit, míg a második részben a szeletek segítségével adjuk meg a program komponensei között meghatározott függőségi kapcsolatok pontosságát.

Nem meglepő, hogy a programszeletelés a szoftverfejlesztés egyik kimagasló területévé nőtte ki magát az elmúlt évtizedekben, hiszen számtalan alkalmazás építhető az eredményeire, számtalan alkalmazás válhat hatékonyabbá általa. Ilyenek a hibakeresés, tesztelés, programmegértés, karbantartás. Míg a magasszintű nyelvek szeletelésének szakirodalma igen gazdag, addig binárisok szeletelésével viszonylag kevesen foglalkoznak annak ellenére, hogy ennek felhasználási területei akár túl is mutathatnak a magasszintű nyelveknél megismert felhasználási területeken. Az értekezés első részében binárisok függőségi gráfjaira épített interprocedurális szeletelésének lépéseit, problémáit ismertettük. A bemutatott eljárást gyakorlati megvalósításunk segítségével ki is értékeltük, illetve tovább pontosítottuk.

Egy lehetséges felhasználási területe a programszeletelésnek a program komponensei közötti függőségek vizsgálata, aminek fontossága megkérdőjelezhetetlen. Ahhoz, hogy

egy szoftver rendszert valójában megérthessünk, megismerjünk, és tudjuk, hogy adott szituációban mit várhatunk el tőle, fontos tudnunk, hogy annak alkotóelemei hogyan, mi módon kapcsolódnak egymáshoz, hatnak egymásra. Habár a függőségek vizsgálatának számos absztrakciós szintje lehet – kezdve a programrendszert leíró modellektől –, a valódi függőségeket és kapcsolatokat a megvalósított rendszer implementációja hordozza magában. Persze attól függően, hogy milyen szinten szeretnénk meghatározni a programban rejlő függőségeket, más és más módszereket alkalmazhatunk. Az értekezés második részében bemutatott eljárásunk célja magasszintű nyelvek eljárásai, illetve osztályai közötti függőségeknek a meghatározása, illetve ezen függőségek vizsgálata. Mivel a programszeletelés utasításszintű kapcsolatok, azaz elemibb szintű függőségek feltárására képes, így arra használjuk fel, hogy eredményével validáljuk a saját módszerünk pontosságát, illetve hatékonyságát.

Bináris programok szeletelése

(A kapcsolódó eredmények a [38; 39] publikációkban találhatóak.)

Habár binárisok szeletelése hasonlóan széles körben felhasználható lehet, mint a magasszintű nyelvek szeletelése, kutatásaink kezdetén nem találkoztunk olyan munkákkal, amely bináris programok interprocedurális szeletelésének teljes folyamatát leírják. Binárisok intraprocedurális szeletelésével ugyan foglalkoznak többen is [22; 42], de az interprocedurális szeletelésre csupán csak javasolják, hogy azt függőségi gráfok segítségével kellene meghatározni [10]. Mivel a bináris programok számos olyan tulajdonsággal rendelkeznek, amikkel a magasszintű nyelveknél nem találkozunk, így a függőségi gráfok felépítésének magasszintű nyelveknél megismert lépései nem adaptálhatóak egy az egyben bináris programok esetén. Már a bináris program vezérlési folyam gráfjának építésekor is számos olyan architektúrától függő nehézséggel kell szembenéznünk, amely problémák magasszintű nyelvek esetén nem is jelentkeznek [24; 37]. A további függőségek és függőségi gráfok meghatározásai pedig újabb problémák megoldását igénylik.

Az értekezés első részében megadjuk milyen problémákkal találkozunk a bináris programok függőségi gráfjainak építése során, és ezen problémákra megoldási javaslatokat is adunk. Ahhoz, hogy minél pontosabb statikus szeleteket kapjunk, a függőségi gráfokat úgy kell felépítenünk, hogy azok a programban rejlő függőségeket a lehető legpontosabban tükrözzék. Statikus elemzés révén a függőségek ábrázolásának konzervatívnak kell lennie, azaz nem hagyhatunk el olyan függőségi élt a gráfból, amiről nem tudjuk egyértelműen eldönteni, hogy mögötte valóban létezik-e függőség, vagy sem. Köszönhetően annak, hogy a bináris programok az adatokat a memóriában tárolják, és az egyes utasítások csak pár regiszter segítségével tudják azokat elérni, az általános adatfüggőségi elemzés bináris programok esetében meglehetősen konzervatív. Ezt a konzervatív

adatfolyam elemzést próbáltuk meg pontosítani a függvényekhez kapcsolódó információk heurisztikus vizsgálatával, illetve a memóriahasználat elemzésével. A programszeletelés bizonyos felhasználásaiban, mint amilyen a hibakeresés, nem feltétlenül kell ragaszkodnunk a függőségek konzervatív kezeléséhez. Dinamikus adatokkal pontosítva a hívási gráfot, tovább csökkenthetjük a szeletek méretét, mellyel még hatékonyabbá tehetjük a szeletelés eredményét felhasználó alkalmazásokat.

Valamennyi bemutatott eljárásunkat és pontosításunkat implementáltuk, és kiértékeltünk statikusan linkelt ARM binárisok segítségével. A hagyományos szeletelési technikával a programmérethez viszonyítva átlagosan 52%-os szelet méretet értünk el, amit a statikus javításainkkal további 1-4%-kal tudtunk javítani. A dinamikus pontosításokkal még ha el is veszítettük a szeletek konzervatív tulajdonságát, a szelet méreteket drámaian tudtuk tovább csökkenteni. A legkiugróbb esetben a program eredeti szelet méretét majdnem harmadára redukáltuk.

Statikus Execute After és Statikus Execute Before relációk

(A kapcsolódó eredmények a [12; 33; 34] publikációkban találhatóak.)

Az értekezés második részében a szeletelés egy felhasználási területével, a programban rejlő függőségek vizsgálatával foglalkoztunk. Habár a programszeletelés potenciálisan alkalmas arra, hogy a programban, a program komponensei között rejlő függőségeket feltárja, nagyobb rendszerek esetén a szeletelés hagyományos technikái nem alkalmazhatóak a felhasznált programreprezentációk méretei miatt, ráadásul sok esetben nincs is szükségünk olyan szintű kapcsolatok feltérképezésére, mint amit a szeletelés biztosít.

Számos alkalmazásban a program eljárásai közötti függőségeket csupán a hívási gráffal [18], míg az osztályok közötti függőségeket valamely csatolási metrika által határozzák meg [19; 63], amely módszerek nem biztonságosak, hiszen könnyen megmutatható, hogy nem ismernek fel olyan függőségeket, amik jelen vannak a programban.

A Statikus Execute After és a Statikus Execute Before relációk bevezetésével célunk egy olyan eszköz megadása volt, amely az előző módszerekhez hasonlóan egyszerűen meghatározható adatok alapján adja meg, hogy a program mely eljárásai, illetve osztályai lehetnek függőségi relációban egymással, de amely módszer ennek ellenére konzervatív, azaz nem veszít el létező kapcsolatot. A program egy redukált reprezentációját, az ICCFG gráfot *(Interprocedural Component Control Flow Graph)*, definiáltuk annak érdekében, hogy ezeket a relációkat egyszerűen meghatározhassuk. Több lehetséges módszert is adtunk ezeknek a relációknak a meghatározására, amely módszerek valamilyen tekintetben

a szélsőséges helyzeteket kezelik le. Az egyik módszer esetében csupán az ICCFG gráfra, és annak bejárására támaszkodunk egy-egy eljárás kapcsolatainak feltérképezésekor. A másik megvalósításban pedig valamennyi eljárás kapcsolatait egyszerre határozzuk meg az ICCFG csomópontjainak egyszeri érintésével.

A programszeletelés eredményeivel összevetve a SEA és SEB relációkat kijelenthetjük, hogy eljárás, illetve osztály szinten ezen relációk meghatározása alkalmas a programban rejlő függőségek közelítésére. További gyakorlati összehasonlításainkkal azt is megmutattuk, hogy a korábban említett, csupán hívási gráfra és objektumorientált metrikákra épülő függőségi elemzések önmagukban azonban nem elegendőek a függőségek meghatározásakor.

Konklúzió

Megadtuk a bináris programok interprocedurális szeletelési folyamatának teljes leírását. Összegyűjtöttük a szeletelési folyamat egyes lépései során jelentkező problémákat, amely problémákra gyakorlati megoldásokat adtunk. A bemutatott módszer implementálásával és gyakorlati kiértékelésével igazoltuk a bináris programok statikus szeletelésének használhatóságát, amit statikus és dinamikus adatokkal való pontosításokkal tovább fokoztunk.

Magasszintű nyelvek függőségeinek eljárás és osztály szintű vizsgálatához bevezettünk egy olyan módszert, amely még nagy programrendszerek esetén is használható. Ez annak köszönhető, hogy meghatározásához elegendő egy olyan tömör programreprezentáció, amely a program hívási gráfját csak minimális információval egészíti ki, amely információ megadja az egyes hívási pontok közötti vezérlési információkat egy-egy eljáráson belül. A megadott módszer pontosságát és hatékonyságát a programszeletelés segítségével igazoltuk.

C. Függelék

Summary in English

"All our science, measured against reality, is primitive and childlike – and yet it is the most precious thing we have."

— Albert Einstein

Introduction

The main topic of the dissertation is program slicing and, indirectly, an investigation of the dependencies of program components. In the first part of the dissertation we compute the program slices of binary executables with the help of dependence graphs, then in the second part we test the dependency model introduced by us with the help of program slicing results. In the first part, our goal is to compute the program slices of binary executables with dependence graphs, while in the second part our goal is to define the precision of dependencies identified between the program components with the help of program slicing.

It is not surprising that program slicing is one of the most important topics of software maintenance, since many applications based on it can become more efficient. These applications include debugging, testing, program comprehension and program maintenance. While the literature about the program slicing of high level languages is rather rich, comparatively little attention has been paid to the slicing of binary executables. The lack of existing solutions is really hard to understand since the application domain for slicing binaries is similar to that for slicing high-level languages. In the first part of the dissertation, we present the steps of the interprocedural program slicing of binary executables and its associated problems. We implemented, evaluated and further improved our solution.

One potential application area of program slicing is the identification of dependencies

among the program components. In order to understand a piece of software and learn about its detailed behavior in a particular situation, we have to map the connections and dependencies among its components. Although the dependencies have many levels of abstraction – starting with the model level, which describes the system –, the effective dependencies are contained in the implementation of the system. Based on the level of computation, we can use different methods. In the second part of the thesis, we present a method which finds and investigates the dependencies among the classes and procedures of the program being analyzed. Since program slicing identifies relations on statement level, which is more fine grained level, so we can use its results to prove the precision and effectiveness of our method.

Program slicing of binary executables

(Results related to the author can be found in [38; 39].)

Despite the fact that the application domain of the program slicing of binary executables may be the same as that for the slicing of high level languages, we did not find any existing solutions for the interprocedural slicing of binaries at the beginning of our studies. Although some papers describe the intraprocedural slicing of binaries [22; 42], only one of them suggests the slicing of binaries with program dependence graphs [10]. As binaries have many features which are not present in high level languages, the steps of the building of the dependence graphs of high level languages cannot be applied in the same way. We had to face many difficulties when determining the control flow graphs of the binary executables, which are not present in the case of high level languages [24; 37]. The determination of the data-, control- and system dependence graphs also require many new solutions.

In the first part of the thesis, we describe the problems of building the dependence graphs of binary executables, and then provide solutions for them. The more precise the computed dependence graphs are, the more precise the computed static program slices will be. When performing a static analysis, the computed dependencies must be conservative. This means that we cannot eliminate a dependence edge from the graph if we do not know whether there is any dependence between its nodes or not. Data dependence computation for binary executables is very conservative due to the fact that the statements of the binary programs can access the data only via a few registers. We attempted to improve this conservative data flow analysis by using some architecture specific information and the analysis of memory use. In some applications of program slices (e.g. debugging), the conservative handling of the dependencies is not so important. Improving the call graph with dynamic information, we can further reduce the sizes of the program slices and hence improve the effectiveness of the applications

based on the results of slicing.

We implemented a slicer and our improvements for statically linked ARM binaries. Using the conservative approach we achieved an interprocedural slice size of 52% on average, which was further reduced by 1-4% by applying the static improvements. Although we lost the conservative property of static slicing with the dynamic improvements of the call graph, the slice sizes were dramatically reduced. In the best case, the original slice size was reduced to one third.

Static Execute After and Static Execute Before relations

(Results related to the author can be found in [12; 33; 34].)

In the second part of the thesis, we shall focus on one application area of program slicing, namely that of determining the dependencies of program components. Although program slicing is potentially suitable for finding the dependencies of the program and the program components, with bigger software systems the usual techniques of program slicing fail due to the huge sizes of their program representations. In addition, in many cases determining the dependencies on statement level is not really necessary.

Many applications determine the dependencies among the procedures just via the call graph [18], while the dependencies among the classes are identified by utilizing coupling metrics [19; 63]. Unfortunately these methods are not safe. It is not hard to prove that in some cases they do not recognize the existing dependencies.

By introducing the Static Execute After and Static Execute Before relations, our intention was to provide a method for determining the dependencies among procedures and classes of programs. This method is conservative and does not ignore any existing dependencies. We shall present the ICCFG graph *(Interprocedural Component Control Flow Graph)*, which is a reduced graph representation for finding these relations. We also provide several alternative methods for computing them. These methods can handle several extreme cases. One method requires just the traversal of the ICCFG and only identifies the relations of one procedure, while another stores and finds the relations of every procedures simultaneously.

When comparing the SEA and SEB relations with the relations computed by program slicing we can say that the determination of these relations at the procedure level and the class level is suitable for approximating dependencies of a given program. We can show by making additional comparisons that the call graph itself or any object-oriented metrics is not sufficient to determine the existing dependencies of a program.

Conclusions

We presented the steps of the interprocedural program slicing of binary executables and discussed some of the problems associated with it. We provided solutions for some of them. By implementing the prototype version of our method, we investigated the usefulness of interprocedural program slicing and its static and dynamic improvements.

In order to examine the procedure level and class level dependencies that exist in high level languages, we introduced a method which was suitable for large programs owing to the compact program representation used. This program representation completes the program call graph just with some additional information that tells us about the control flow data inside a given procedures. The precision and effectiveness of our novel method was tested by the relations computed by program slicing.

Irodalomjegyzék

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM.

[2] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

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

[4] David Francis Bacon. *Fast and effective optimization of statically typed object-oriented languages*. PhD thesis, EECS Department, University of California, Berkeley, 1997. Chair-Susan L. Graham.

[5] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 167–175, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer-Verlag, 2004.

[7] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging*, pages 206–222, 1993.

[8] Leeann Bent, Darren C. Atkinson, and William G. Griswold. A qualitative study of two whole-program slicers for C.

[9] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[10] J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 184–189, Washington, DC, USA, 1999. IEEE Computer Society.

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

[12] Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, pages 295–304. IEEE Computer Society, October 2007.

[13] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 105–113, March 2001.

[14] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[15] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pages 44–53. IEEE Computer Society, September 2003.

[16] David Binkley and Mark Harman. A survey of empirical results on program slicing. In *Advances in Computers*, pages 105–178, 2004.

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

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

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

[20] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20:1265–1296, 1998.

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

[22] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 188–195, October 1997.

[23] Andrea de Lucia. Program slicing: Methods and applications. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:0144, 2001.

[24] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[25] 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, October 2002.

[26] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392, New York, NY, USA, 1995. ACM.

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

[28] Loukas Georgiadis. *Linear-time algorithms for dominators and related problems*. PhD thesis, Princeton, NJ, USA, 2005. Adviser-Robert E. Tarjan.

[29] GrammaTech's CodeSurfer. `http://www.grammatech.com/products/codesurfer`.

[30] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Software Focus*, pages 70–79. IEEE Computer Society Press, 1997.

[31] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

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

[33] Judit Jász. Static execute after algorithms as alternatives for impact analysis. *Peryodica Politechnica*, page Submitted paper, Budapest, 2009.

[34] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*, pages 137–146. IEEE Computer Society, October 2008.

[35] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

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

[37] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 46–55, New York, NY, USA, 2002. ACM.

[38] Ákos Kiss, Judit Jász, and Tibor Gyimóthy. Using dynamic information in interprocedural static slicing of binary executables. *Software Quality Journal*, 13(3):227–245, September 2005.

[39] 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, September 2003.

[40] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(2):155–163, 1988.

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

[42] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 30(6):291–300, June 1995.

[43] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 430 – 441, November 2003.

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

[45] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. International Symposium on Microarchitecture*, pages 330–335, 1997.

[46] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[47] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification Version 6.0, February 1999.

[48] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 71–80, New York, NY, USA, 2002. ACM.

[49] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

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

[51] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[52] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[53] Jeff Russell. Program slicing literature survey. 2001.

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

[55] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[56] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 benchmarks.

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

[58] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[59] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Version 1.2, May 1995.

[60] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, New York, NY, USA, 1991. ACM.

[61] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigen, 1979.

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

[63] F. George Wilkie and Barbara A. Kitchenham. Coupling measures and change ripples in C++ application software. *Journal of Systems and Software*, 52(2–3):157–164, 2000.

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