Függőség alapú statikus programszeletelés és közelítései

Ph.D. értekezés tézisei

Jász Judit
Témavezető: Dr. Gyimóthy Tibor

Szegedi Tudományegyetem
Természettudományi és Informatikai Kar
Informatika Doktori Iskola

Szeged, 2009

Bevezetés

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 önálló programként 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 [26].

Ahogy nőtt a programszeleteket meghatározó algoritmusoknak a száma, különböző módosított definícióit javasolták [5; 6; 18; 25]. A legfőbb oka ezeknek az eltéréseknek 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 programszeletelés módszerét 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 végrehajtásai megelőzhették 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 a 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* szeletelésről.

Az értekezésben a programszeletelés statikus változatával, és annak egy lehetséges alkalmazásával, a programon belüli függőségek vizsgálatával foglalkozunk. Míg az első részben a függőségi gráfokat, függőségi kapcsolatokat használjuk arra, hogy segítségükkel megadjuk bináris programok statikus interprocedurális szeleteit, addig a második részben a szeletelést használjuk fel arra, hogy validáljuk a szeleteléstől függetlenül meghatározott függőségi kapcsolatokat. Az értekezés a következő téziseket tartalmazza témaköreinek megfelelően két csoportba osztva.

I/1 **Bináris programok függőségi gráfjainak előállítási problémái és megoldásai, függőség alapú szeletelés adaptálása bináris programokra.**

I/2 **Módosított szeletelési algoritmusok kidolgozása a bináris programok szeletelésének pontosítására.**

I/3 **Bináris programok statikus szeletelésének gyakorlati kiértékelése.**

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.**

II/2 **A SEA és SEB relációk gyakorlati összevetése a programszeleteléssel.**

II/3 **Rejtett függőségek objektumorientált programokban; a SEA és SEB relációk gyakorlati vizsgálata objektumorientált programokban.**

A következő fejezetekben röviden összefoglalom ezeket a téziseket. Minden fejezet végén külön kiemelve a saját hozzájárulásomat az eredményekhez.

Bináris programok szeletelése

Számos 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.

Binárisok intraprocedurális szeletelésével ugyan foglalkoznak többen is [9; 20], de az interprocedurális szeletelésre csupán csak javasolják, hogy azt függőségi gráfok segítségével kellene meghatározni [2]. 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.

I/1. Bináris programok függőségi gráfjainak előállítási problémái és megoldásai, függőség alapú szeletelés adaptálása bináris programokra

Egy lehetséges módszer a programok interprocedurális szeleteinek meghatározására, ha felépítjük a program megfelelő függőségi gráfjait, majd annak egy megfelelő bejárásával megadjuk a kívánt szeleteket [12]. Bináris programok függőségi gráfjainak előállításakor számos olyan problémával találkozunk, amelyek nem fordulnak elő magasszintű nyelveknél.

Mivel a vezérlési folyam gráf *(Control Flow Graph – CFG)* felépítése számos egyéb alkalmazásban (kódelemzés, kódgenerálás, kódtömörítés) is szerepet kap, így binárisok esetében is találunk számos munkát, melyek ezzel foglalkoznak [10; 15]. Attól függően, hogy éppen milyen architektúrára fordított binárissal dolgozunk, más és más problémák merülhetnek fel. Általánosan azonban igaz az, hogy a vezérlési folyam gráf előállítása bináris programok esetében nem csupán az egyes utasítások közötti vezérlési információk haladásának meghatározását jelentik, de magukban foglalják az egyes utasítások, függvények határainak kijelölését is.

A vezérlési információk előállítása után az adatfüggőségi gráf *(Data Dependence Graph – DDG)* és vezérlésfüggőségi gráf *(Control Dependence Graph – CDG)* együttesen határozzák meg valamennyi eljárásnak a programfüggőségi gráfját *(Program Dependence Graph – PDG)*. A megfelelő élekkel kiegészítve a PDG-t megkapjuk a rendszerfüggőségi gráfot *(System Dependence Graph – SDG)*, amelyet közvetlenül felhasználunk az interprocedurális szeletek kiszámításához.

A vezérlési folyam információk alapján a vezérlésfüggőségek megadása egyszerűen adódik bináris programok esetében is, bár az átlapolásokat és kereszthivatkozásokat kezelnünk kell. Mivel a bináris programok esetén az egyes függvények közötti váltás nemcsak függvényhívások révén valósulhat meg, így a dominancia relációk meghatározásakor az egyes függvényeket alkotó programpontok körét ki kell terjesztenünk az összes olyan utasításra, amelyeket egy adott függvény belépési pontjából érhetünk el valódi függvényhívás nélkül.

Az adat függőségek pontos meghatározása a legnehezebb feladat a binárisok esetében. Ellentétben a magasszintű nyelvek utasításaival, a bináris programok utasításai regisztereket, jelzőbiteket és a memória tartalmát írják, olvassák. Az adatfüggőségi elemzés első lépése tehát annak meghatározása, hogy az egyes utasítások mely regisztereket, jelzőbiteket olvassák (írják), illetve meg kell határoznunk azt is, hogy a memóriához miként férnek hozzá. Konzervatív megközelítésünkben csupán annyit tettünk, hogy eldöntöttük, adott utasítás olvassa-e, vagy írja-e a memóriát, míg a pontosabb

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

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

1. ábra. Az U_f és D_f halmazok meghatározása, ahol U_f és D_f halmazok az f függvény által használt és definiált elemek halmazai, I_f az f függvény utasításainak, C_f az f függvény által hívott függvényeknek a halmaza, míg u_j és d_j az indexszel jelölt utasítás által használt és definiált elemek halmaza.

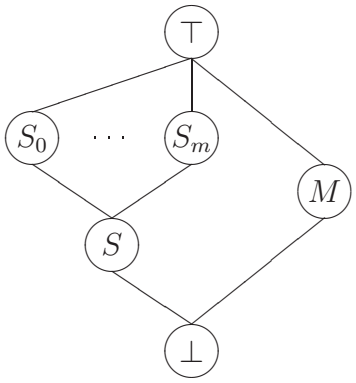
megvalósításokban azt is megpróbáltuk behatárolni, hogy adott esetben a memória mely része lehet érintett az adott utasítás által.

Mivel bináris programok esetében az egyes függvényeknek expliciten nem adottak a paraméterlistái, az egyes függvények által használt, illetve definiált regisztereket, jelzőbiteket, illetve memória helyeket nekünk kell meghatározni. Erre az 1. ábrán bemutatott fixpont iterációs módszert vezettünk be, amely meghatározza minden egyes f függvényre az általa használt és definiált elemek U_f és D_f halmazait.

A bináris programot reprezentáló gráfot kiegészítve a megfelelő regiszter, jelzőbit és memória használatot jelképező csomópontokkal, az adatfüggőségi élek már meghatározhatóak a vezérlési folyam gráf élein haladva. Az interprocedurális szeleteléshez szükséges összegző élek ezután szintén megadhatóak a megfelelő algoritmus implementálásával [22].

I/2. Módosított szeletelési algoritmusok kidolgozása a bináris programok szeletelésének pontosítására

Habár a függőségi gráfok 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 össze-



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

2. ábra. Háló és a háló metszet művelete a memóriahasználatok pontosításához

gyűjtött információkat is felhasználunk a gráfok felépítésekor.

Az eredeti algoritmus mellet megadtunk két statikus és két dinamikus módszert, amelyekkel az eddig megadott függőségi gráfokat pontosítottuk. A statikus módszerek konzervatívak, azaz általuk nem veszítünk el valódi függőségeket. A dinamikus módszerekről ugyanez nem mondható el, mivel azonban a szeletelésnek elképzelhetőek olyan felhasználási területei, amik nem is követelik meg a függőségeknek ezt a konzervatív kezelését, így ezek a módszerek is használhatóak lehetnek.

Az első statikus módszernél azt használtuk ki, hogy a jelenleg használt architektúrák legtöbbjén léteznek olyan függvényhívási megszorítások, amik előírják, hogy az egyes függvényeknek mely regisztereket kell változatlanul hagyniuk. Az elmentett és visszaállított regiszterek kiemelésével csökkenteni tudjuk az egyes függvények által módosított regiszterek számát, így közvetve az összegző élek számát, amely által csökken a szelet méretek nagysága is.

A második statikus módszerben az adat függőség konzervatív kezelését próbáltuk finomítani. Mivel a bináris utasítások által használt regiszterek száma véges, így az utasítások a regiszterek tartalmának pillanatnyi eltárolását sokszor a memória egy meghatározott területének, a veremnek a segítségével oldják meg. Azzal, hogy minden utasítás előtt, és minden utasítás végrehajtása után jellemezzük annak regisztereit aszerint, hogy az általuk tartalmazott adatok honnan származhatnak, pontosíthatjuk az adatfüggőségi elemzésünket. Egy hálót bevezetve a regisztertartalom jellemzésére mondhatjuk, hogy adott érték nem függ a memóriától ($\top$), vermen kívüli adatot tartalmaz (M), a verem egy meghatározott pontjáról olvasott értéket tartalmaz (S_i), a verem egy nem definiált pontjának értékét tartalmazza (S), vagy legrosszabb esetben azt mondjuk, hogy statikusan nem eldönthető, hogy a regiszter hivatkozik-e verem

tartalomra, vagy sem ($\perp$). A hálót és a hálón értelmezett metszet műveletet a 2. ábra mutatja be. A megfelelő inicializálás után az egyes utasítások regisztereit jellemző hálóelemeket egy újabb fixpont iterációs algoritmus adja, amelyet az értekezésben mutatunk be részletesen.

Bináris programok esetében, különösen a nagyobb programoknál, számos esetben találunk statikusan fel nem oldott függvényhívásokat. Ezek jelenléte miatt így itt is számos olyan függőség kerül be a gráfba, amely gyakorlatilag nem valósul meg. A dinamikus javításokkal az indirekt függvényhívások által behozott függőségeket próbáljuk pontosítani. Ezt úgy érhetjük el, hogy a statikusan fel nem oldott indirekt függvényhívások címének kigyűjtése után reprezentatív bemenetekre futtatjuk a vizsgált bináris programot egy vezérelhető eszköz segítségével, amely lehet akár egy szoftver emulátor, vagy egy valós gép egy megfelelő nyomkövető felülettel, majd az előbb meghatározott címeknél, mint töréspontoknál a regiszterek aktuális értékeit kigyűjtjük.

A statikusan fel nem oldott indirekt függvényhívások célját pontosíthatjuk a kigyűjtött információk segítségével. Az általunk megadott két módszer abban különbözik egymástól, hogy míg az egyiknél csak a futtatások által érintett indirekt hívások célpontjait módosítjuk csak, addig a másiknál valamennyi indirekt függvényhívási célt eltöröljük, és csak azokon a pontokon határozzuk meg a hívás célját, amely hívásra a vezérlés rákerült valamelyik teszteset futtatásakor.

I/3. Bináris programok statikus szeletelésének gyakorlati kiértékelése.

Az ismertetett eljárásokat statikusan linkelt ARM programok elemzéséhez implementáltuk, majd a SPEC CINT2000 [24] és Media Bench benchmark suit [21] programjain értékeltük ki. A kiválasztott programokat a Thumb utasításkészletű ARM7T processzorra fordítottuk a Texas Instruments TMS470R1x Optimalizált C fordítójának 1.27e verziójával. A dinamikus adatokkal való pontosításhoz a vizsgált programokat a Texas Instruments TMS470R1x C Source Debugger emulátorával futtattuk.

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. A függőségi gráfok statikus pontosításával 1% - 4 %-os szelet méret csökkenést sikerült elérni. A relatív kis mértékű javulás a memória-elérések még így is konzervatív kezelésének, és a nagyszámú indirekt függvényhívásnak köszönhető.

A dinamikus javítások rámutatnak arra, hogy nagy az összefüggés a hívási élek számának csökkenése és a kapott szelet méretek között. Azokban az esetekben, ahol az indirekt hívások száma meghatározó volt, és ahol így a dinamikus módszerekkel na-

gyobb mértékben tudtuk csökkenteni a hívási élek számát, ott a kapott szelet méretek is jóval kisebbek lettek az eredeti szeletelési algoritmus eredményeihez képest.

A szerző hozzájárulása az eredményekhez

Az első részben bemutatott munkánkat az a tény motiválta, hogy bár bináris programok szeletelésének is számos felhasználási módja lehet, a szakirodalom ezzel a témával mégsem foglalkozik kellő részletességgel. Bináris programok iterprocedurális szeletelését függőségi gráfjai segítségével adtuk meg, ahol a vezérlésfüggőségi, adatfüggőségi és a rendszerfüggőségi gráfok felépítésének megadása a szerző önálló munkája. A szerző önálló eredménye még az adatfüggőségi háló révén megvalósított pontosított adatfüggőségi elemzés. A függőségi gráfok további pontosítási technikáinak megtervezése a szerzőtársakkal oszthatatlan eredménynek minősül, mint ahogy a mérések megtervezése, és a mérési eredmények kiértékelése is. A vezérlési folyam gráf előállításának implementálásán kívül valamennyi függőségi gráf gyakorlatban való megvalósítása, illetve az ismertetett módszerek, pontosítások implementálása a szerző munkája.

Statikus Execute After és Statikus Execute Before relációk

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 [7], míg az osztályok közötti függőségeket valamely csatolási metrika által határozzák meg [8; 27], amely módszerek nem biztonságosak, hiszen könnyen megmutatható, hogy nem ismernek fel olyan függőségeket, amik jelen vannak a programban.

Az értekezés második részében egy olyan technikát mutattunk be, amely egyszerűsége mellett rendelkezik azzal a tulajdonsággal, hogy a programban rejlő szemantikus függőségeket konzervatív módon közelítse. A programszeleteléssel összevetve az általunk bevezetett módszer eredményét azt is elmondhatjuk, hogy ez a közelítés eljárás szinten, illetve magasabb szinten relatív nagy pontossággal rendelkezik.

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.

Az alapötletet, amellyel a program eljárásai közötti függőségeket közelítettük, Appiwattanapong és társai által bevezetett *Execute After (EA)* reláció adta [1]. 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. 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.

Apiwattanapong és társai [1], valamint Beszédes és társai [3] 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 csak az érdekel minket, hogy egy adott eljárás hogyan viselkedik a többi eljárással.

Ahhoz, hogy meghatározhassuk egy program SEA relációinak a halmazát, egy erre megfelelő programreprezentációt kell előállítanunk. A hagyományos hívási gráf nem elegendő [23], hiszen ez semmit nem mond az egy eljáráson belül megvalósuló eljáráshívások sorrendjéről. Másfelől az interprocedurális vezérlési folyam gráf *(Interprocedural Control Flow Graph – ICFG)* [19] túl sok olyan információt tartalmaz, amely már nem köthető az eljárás hívásokhoz, így számunkra feleslegesek.

Az általunk definiált interprocedurális komponens vezérlési folyam gráfban *(Interprocedural Component Control Flow Graph – ICCFG)* minden eljárást egy-egy komponens vezérlési folyam gráf *(Component Control Flow Graph – CCFG)* reprezentál, amelyet egy belépési csomópont, és komponens csomópontok építenek fel. 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 összekötjük, kihagyva azon csomópontokat, amelyek nem tartalmaznak hívási helyeket. Az egyes CCFG gráfokat hívási élek kötik össze, amely élek a hívási helyeket reprezentáló csomópontokat kötik össze a megfelelő eljárások belépési csomópontjával.

A SEA és SEB relációk meghatározására több lehetséges módszert is adtunk, amely módszerek valamilyen tekintetben a szélsőséges helyzeteket kezelik le. Az egyik módszer esetében csupán az ICCFG gráfra, és annak bejárására támaszkodtunk egy-egy eljárás kapcsolatainak feltérképezésekor. A másik megvalósításban pedig valamennyi eljárás kapcsolatait egyszerre határoztuk meg az ICCFG csomópontjainak egyszeri érintésével.

II/2. A SEA és SEB relációk gyakorlati összevetése a programszeleteléssel

Ahhoz, hogy belássuk, hogy a SEA és SEB relációk alkalmasak az eljárások között fellépő szemantikus függőségek közelítésére, a programszeletelést használtuk fel. Méréseink során a meghatározott SEA, illetve SEB relációkat egy erre alkalmas szeletelő eszköz, a CodeSurfer [11] eredményeivel vetettük össze. A SEA relációkat értelemszerűen 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 összehasonlításhoz a szelethez viszonyított pontosság és fedettség értékeket használtuk. 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 valamely kritériumából indított programszeletet megkaphatjuk a rendszerfüggőségi gráf éleinek megfelelő bejárásával [12]. 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. Ennek a hibának az okaival és hatásával az értekezés függelékében részletesen foglalkozunk. A hibának köszönhetően a gyakorlati mérésekben csak a SEB halmazoknak és a hátrahaladó szeletek által meghatározott kapcsolatoknak a viszonyát vizsgáltuk. Huszonkilenc C program esetén vizsgáltuk a pontosság értékek alakulását eljárás szinten. Az esetek többségében a pontosság érték elérte a 90%-ot. Ez a magas pontosság érték annak is köszönhető, hogy az eltérés a szeletelés által meghatározott kapcsolatok között, és a SEA (SEB) relációk által meghatározott kapcsolatok között átlagosan 4% körül adódott.

Mivel mind a szeletelés, mind a SEB relációk meghatározása tulajdonképpen egy-egy gráfbejárási algoritmus eredményeként adódnak, nem mellékes szempont, hogy mekkora az adott gráf mérete, amin ezeket az algoritmusokat futtatjuk. Ahhoz, hogy a módszerünk hatékonyságát is megmutassuk, nagyobb C++ rendszerek megfelelő függőségi gráfjait állítottuk elő, és ezek méretét hasonlítottuk össze. Bár a gráfok méreteinek különbözősége önmagában is meggyőzően igazolja, hogy a SEA (SEB) relációk meghatározása sokkal kisebb erőforrást emészt fel, érdemes kiemelni azt a tényt, hogy míg a SEA (SEB) relációk meghatározásához szükséges ICCFG gráf előállítása a milliós sormérettel rendelkező `mozilla` program esetén gond nélkül megtörtént, addig a szeleteléshez szükséges SDG előállítása már erőforrás korlátokba ütközött.

II/3. Rejtett függőségek objektumorientált programokban; a SEA és SEB relációk gyakorlati vizsgálata objektumorientált programokban

Amikor rejtett függőségekről beszélünk, azokra a függőségi kapcsolatokra gondolunk, amelyek expliciten nem olvashatóak ki a program forráskódjából, amelyek meghatározása egy alaposabb elemzés eredménye lehet. A célunk a rejtett függőségek vizsgálatakor az volt, hogy objektumorientált programok osztályai közötti összes lehetséges kapcsolatot összegyűjtsük, ne csak az expliciten meghatározhatóakat. Ezt a szándékunkat az teszi 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.

Ahhoz, hogy osztályok közötti összes függőséget, illetve kapcsolatot megadjuk, az eredetileg eljárások között értelmezett SEA és SEB relációkat terjesztettük ki osztály szintre. 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 ez a technika, illetve a globális változók és osztályok közötti kapcsolatokat sem veszi észre. A globális változók és a tagváltozók közvetlen elérésével létrejött kapcsolatok láthatatlanok maradnak az algoritmus számára, hiszen az ezeknek megfelelő csomópontok hiányoznak az ICCFG gráfból. Az értekezésben bemutatjuk azokat a lehetséges program transzformációkat, amelyek által az ilyen szituációk is elérhetővé válnak számunkra.

A gyakorlati mérések, melyekben C++ és Java programokat vizsgáltunk, igazolták, hogy az expliciten megjelenő függőségek száma jóval kevesebb, mint a rejtett függőségek száma. Így ez azt jelenti, hogy önmagukban az ezeken a metrikákon alapuló módszerek nem adnak megbízható eredményt olyan alkalmazásokban, amelyekben ezek segítségével keresik az egymástól függő osztályokat. Vizsgálva az egy osztályhoz tartozó függőségek számát az is megfigyelhető, hogy jellemzően előforduló az az eset, amikor a létrejövő függőségek klasztereket alkotnak. A klaszterek az osztályoknak olyan csoportjai, amelyek elemei nemcsak hogy megegyező számú függőséggel rendelkeznek, de ezek a függőségek nagyjából meg is egyeznek. A rejtett függőségek magas értéke, illetve a nagy klaszterek jelenléte veszélyes, hiszen a kód biztonságos karbantartása ezek mellett

nehéz. Mivel a rejtett függőségek és az egy osztályhoz tartozó SEA (SEB) kapcsolatok száma erősen korrelálnak egymással, így ez utóbbi relációk meghatározása segíthet a rendszer nehezebben karbantartható osztályainak kiszűrésében, mint ahogyan segíthet a nagyobb klaszterek megtalálásában és felszámolásában is.

A szerző hozzájárulása az eredményekhez

A második részben bemutatott kutatásunk célja egy olyan eszköz megadása volt, amely a program eljárásai, illetve osztályai közötti függőségeket képes közelíteni hatékonyan, és pontosan. Az erre a célra bevezetett SEA, illetve SEB relációk, valamint az ICCFG gráf definiálása a szerzőtársakkal oszthatatlan eredményeknek minősülnek. Az algoritmusok, amelyekkel meghatározhatóak ezek a relációk, illetve az algoritmusokhoz köthető mérési eredmények a szerző önálló eredményei. A SEA és SEB relációk és a programszeletelés összevetésének gyakorlati megtervezése és kivitelezése a szerző önálló munkája. A bevezetett relációk objektumorientált metrikákkal való összevetésének gyakorlati megtervezése a kapcsolódó publikáció szerzőivel közös eredmény, míg a C/C++ programokhoz kapcsolódó mérések kivitelezése, valamint a SEA (SEB) relációk gyakorlati meghatározása a szerző munkája.

Összefoglalás

Az értekezés első részében bináris programok statikus szeletelésének problémáival foglalkoztunk. Megadtunk egy konzervatív, függőségi gráfok bejárásán alapuló szeletelési megoldást. A megadott algoritmust statikus és dinamikus adatokkal tovább javítottuk azzal, hogy a függőségi gráfokat pontosítottuk. Valamennyi megoldásunkat implementáltuk, majd az ismertetett módszereket gyakorlati példákon kiértékeltük.

Az értekezés második részben bevezettük a Statikus Execute After és a Statikus Execute Before relációkat, amelyeket magasszintű nyelvek eljárásai és osztályai között lévő kapcsolatok 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,

	[17]	[16]	[4]	[14]	[13]
I/1	•				
I/2	•	•			
I/3	•	•			
II/1			•	•	•
II/2			•	•	•
II/3			•		

1. táblázat. A tézisek és a velük kapcsolatos publikációk viszonya.

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 tegyünk megbízhatóbbakká segítségükkel, mint amilyenek többek között a hatásanalízis, változás propagáció, tesztelés.

Az 1. táblázat összefoglalja az értekezés téziseinek és a kapcsolódó publikációknak a kapcsolatait.

Köszönetnyilvánítás

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.

Jász Judit, 2009. május 8.

Hivatkozások

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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

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

[27] 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.

Társszerzői nyilatkozat

Kijelentem, hogy ismerem *Jász Judit* PhD fokozatra pályázó *Függőség alapú statikus programszeletelés és közelítései* című disszertációját. A disszertációban szereplő és az

- **Ákos Kiss, Judit Jász, and Tibor Gyimóthy.** Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables. *Software Quality Journal*, 13(3):227-245, September 2005.
- **Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy**. Interprocedural Static Slicing of Binary Executables. *In Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 118-127, Amsterdam, The Netherlands, September 26-27, 2003.

cikkekben publikált közös eredményekre vonatkozóan kijelentem, hogy a következő eredményekhez való hozzájárulásunk oszthatatlan:

- Bináris programok hívási (call) gráfjának dinamikus információkkal való pontosítása (4.2. fejezet).
- A bináris programok szeletelése során előállt kísérleti mérési eredmények (5. fejezet).

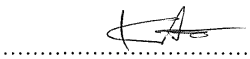
A következő eredményekben a pályázó hozzájárulása volt a meghatározó:

- Bináris programok szeleteléséhez szükséges vezérlési függőségi (control dependence), adatfüggőségi (data dependence) és rendszerfüggőségi (system dependence) gráfok építése. (3.2. és 3.3 fejezetek).
- A statikus pontosításhoz használt háló alapján a pontosított adatfüggőségi analízis kidolgozása (4.1. fejezet).

A következő eredményekben az én hozzájárulásom volt a meghatározó:

- Bináris programok szeletelésének problémáinak feltárása és a szeleteléshez szükséges vezérlési folyam (control flow) gráf építése (3.1. fejezet).
- Bináris programok adatfüggőségi (data dependence) gráfjának statikus pontosításához használt háló kidolgozása. (4.1. fejezet).

Szeged, 2009. május 8.



.....

Kiss Ákos

Társszerzői nyilatkozat

Kijelentem, hogy ismerem *Jász Judit* PhD fokozatra pályázó *Függőség alapú statikus programszeletelés és közelítései* című disszertációját. A disszertációban szereplő és az

- **Judit Jász, Árpád Beszédes, Tibor Gyimóthy and Vaclav Rajlich** Static Execute After/Before as a Replacement of Traditional Software Dependencies. *2008 IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 137-146. Beijing, China, Sept, 2008.
- **Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Vaclav Rajlich:** Computation of Static Execute After Relation with Applications to Software Maintenance. *2007 IEEE International Conference on Software Maintenance (ICSM 2007)*, pp. 295-304. Paris, France, October 2-5, 2007.

cikkekben publikált közös eredményekre vonatkozóan kijelentem, hogy a következő eredményekhez való hozzájárulásunk oszthatatlan:

- A SEA és SEB relációk, illetve a meghatározásukhoz használt ICCFG gráf definiálása (7.1 és 7.2 fejezetek)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének elméleti eredményei (9.1 fejezet)

A következő eredményekben a pályázó hozzájárulása volt a meghatározó:

- Algoritmusok a SEA és SEB relációk meghatározására (7.3 és 7.4 fejezetek)
- A SEA és SEB relációk által meghatározott kapcsolatok és a programszeletelés gyakorlati összehasonlítása (8. fejezet)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének gyakorlati eredményei (9.2 – 9.4. fejezetek)

Szeged, 2009. május 8.

.....

Beszédes Árpád

Társszerzői nyilatkozat

Kijelentem, hogy ismerem *Jász Judit* PhD fokozatra pályázó *Függőség alapú statikus programszeletelés és közelítései* című disszertációját. A disszertációban szereplő és az

- **Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Vaclav Rajlich:** Computation of Static Execute After Relation with Applications to Software Maintenance. *2007 IEEE International Conference on Software Maintenance (ICSM 2007)*, pp. 295-304. Paris, France, October 2-5, 2007.

cikkben publikált közös eredményekre vonatkozóan kijelentem, hogy a következő eredményekhez való hozzájárulásunk oszthatatlan:

- A SEA és SEB relációk, illetve a meghatározásukhoz használt ICCFG gráf definiálása (7.1 és 7.2 fejezetek)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének elméleti eredményei (9.1 fejezet)

A következő eredményekben a pályázó hozzájárulása volt a meghatározó:

- Algoritmusok a SEA és SEB relációk meghatározására (7.3 és 7.4 fejezetek)
- A SEA és SEB relációk által meghatározott kapcsolatok és a programszeletelés gyakorlati összehasonlítása (8. fejezet)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének gyakorlati eredményei (9.2 – 9.4. fejezetek)

Szeged, 2009. május 8.

.....
Gergely Tamás

Társszerzői nyilatkozat

Kijelentem, hogy ismerem *Jász Judit* PhD fokozatra pályázó *Függőség alapú statikus programszeletelés és közelítései* című disszertációját. A disszertációban szereplő és az

- **Judit Jász, Árpád Beszédes, Tibor Gyimóthy and Vaclav Rajlich** Static Execute After/Before as a Replacement of Traditional Software Dependencies. *2008 IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 137-146. Beijing, China, Sept, 2008.
- **Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Vaclav Rajlich:** Computation of Static Execute After Relation with Applications to Software Maintenance. *2007 IEEE International Conference on Software Maintenance (ICSM 2007)*, pp. 295-304. Paris, France, October 2-5, 2007.
- **Ákos Kiss, Judit Jász, and Tibor Gyimóthy.** Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables. *Software Quality Journal*, 13(3):227-245, September 2005.
- **Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy**. Interprocedural Static Slicing of Binary Executables. *In Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 118-127, Amsterdam, The Netherlands, September 26-27, 2003.

cikkekben publikált közös eredményekre vonatkozóan kijelentem, hogy a következő eredményekhez való hozzájárulásunk oszthatatlan:

- Bináris programok hívási (call) gráfjának dinamikus információkkal való pontosítása (4.2. fejezet).
- A bináris programok szeletelése során előállt kísérleti mérési eredmények (5. fejezet).
- A SEA és SEB relációk, illetve a meghatározásukhoz használt ICCFG gráf definiálása (7.1 és 7.2 fejezetek)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének elméleti eredményei (9.1 fejezet)

A következő eredményekben a pályázó hozzájárulása volt a meghatározó:

- Bináris programok szeleteléséhez szükséges vezérlési függőségi (control dependence), adatfüggőségi (data dependence) és rendszerfüggőségi (system dependence) gráfok építése. (3.2. és 3.3 fejezetek).
- A statikus pontosításhoz használt háló alapján a pontosított adatfüggőségi analízis kidolgozása (4.1. fejezet).
- Algoritmusok a SEA és SEB relációk meghatározására (7.3 és 7.4 fejezetek)
- A SEA és SEB relációk által meghatározott kapcsolatok és a programszeletelés gyakorlati összehasonlítása (8. fejezet)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének gyakorlati eredményei (9.2 – 9.4. fejezetek)

Szeged, 2009. május 8.

.....

Gyimóthy Tibor

Társszerzői nyilatkozat

Kijelentem, hogy ismerem *Jász Judit* PhD fokozatra pályázó *Függőség alapú statikus programszeletelés és közelítései* című disszertációját. A disszertációban szereplő és az

- **Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Vaclav Rajlich:** Computation of Static Execute After Relation with Applications to Software Maintenance. *2007 IEEE International Conference on Software Maintenance (ICSM 2007)*, pp. 295-304. Paris, France, October 2-5, 2007.

cikkben publikált közös eredményekre vonatkozóan kijelentem, hogy a következő eredményekben a pályázó hozzájárulása volt a meghatározó:

- Algoritmusok a SEA és SEB relációk meghatározására (7.3 és 7.4 fejezetek)
- A SEA és SEB relációk által meghatározott kapcsolatok és a programszeletelés gyakorlati összehasonlítása (8. fejezet)
- A SEA és SEB relációk objektumorientált metrikákkal való összevetésének gyakorlati eredményei (9.2 – 9.4. fejezetek)

A következő eredményekben az én hozzájárulásom volt a meghatározó:

- A Java programokhoz köthető gyakorlati mérések kivitelezése, a SEA és SEB relációkhoz szükséges információk kinyerése.

Szeged, 2009. május 8.

Tóth Gabriella

Coauthor's declaration

I hereby certify that I am familiar with the thesis of the applicant Judit Jász entitled *Dependence-based Static Program Slicing and its approximations*. Regarding our joint results referred to in this thesis and published in

- **Judit Jász, Árpád Beszédes, Tibor Gyimóthy and Vaclav Rajlich** Static Execute After/Before as a Replacement of Traditional Software Dependencies. *2008 IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 137-146. Beijing, China, Sept, 2008.
- **Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Vaclav Rajlich:** Computation of Static Execute After Relation with Applications to Software Maintenance. *2007 IEEE International Conference on Software Maintenance (ICSM 2007)*, pp. 295-304. Paris, France, October 2-5, 2007.

the following ones were obtained as the result of joint contribution by the applicant and myself:

- The definition of the SEA and SEB relations, and the ICCFG.
- The theoretical results of the comparison of the SEA and SEB relations with object oriented metrics.

The applicant's contribution was prominent in obtaining the following results:

- Determining algorithms to compute the SEA and SEB relations.
- Experimental comparison of the SEA and SEB relations with the results of program slicing.
- Experimental comparison of the SEA and SEB relations with object oriented metrics.

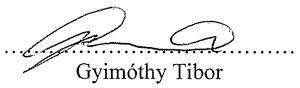
Detroit, 25. May 2009.

……
Václav Rajlich

Témavezetői nyilatkozat

Alulírott, Dr. Gyimóthy Tibor, mint *Jász Judit* doktorjelölt témavezetője kijelentem, hogy a jelölt *Függőség alapú statikus programszeletelés és közelítései* című értekezésében felhasznált eredmények a jelölt saját hozzájárulását tükrözik.

Szeged, 2009. május 8.



Gyimóthy Tibor