Ph.D. értekezés tézisei

Programok Statikus és Dinamikus Analízise

Gergely Tamás

Témavezető
Dr. Gyimóthy Tibor

Informatika Doktori Iskola

Szegedi Tudományegyetem
Informatikai Tanszékcsoport

2010

Bevezetés

A disszertáció témája a program analízis – programelemek közötti relációk meghatározása. A szoftver életciklusában sokféle célra felhasználható, például teszteset szelekcióra, nyomkövetésre, hibakeresésre, programmegértésre, visszatervezésre vagy változás kiterjesztésre.

A programok analízise egy nagy és szerteágazó kutatási terület, de bizonyos szempontok szerint számos területét meg lehet különböztetni egymástól. Ilyen szempont a részletesség és az, hogy a módszer statikus vagy dinamikus. A részletesség adja meg, hogy a relációt magas szintű programelemek (például eljárások, metódusok, osztályok) vagy alacsony szintű elemek (tipikusan forráskódú vagy gépi kódú utasítások) között definiáljuk. A statikus analízis csak statikusan (azaz a program futtatása nélkül) elérhető információkra hagyatkozhat, míg a dinamikus analízis felhasználhat a program futása során gyűjtött információt is.

A *hatásanalízis* egy magas szintű analízis, ami főként statikus technikákat használ, míg a *programszeletelés* egy alacsony szintű analízis statikus és dinamikus alkalmazásokkal. A disszertációban a programanalízisnek ezzel a két területtel foglalkoztunk. Nevezetesen, a statikus és dinamikus hatásanalízis illetve dinamikus szeletelés kutatásában elért eredményeinket mutatjuk be.

Eredményeinket öt tézisben foglaltuk össze.

- I/1. *SEA/SEB* relációk definíciója.
- I/2. *DFC* metrika definíciója, meghatározása és kiértékelése.
- II/1. *d:U* alapú szeletelő algoritmusok meghatározása.
- II/2. *d:U* alapú szeletelő algoritmusok implementációja.
- II/3. *d:U* alapú szeletelő algoritmusok kiértékelése.

I. Magas szintű analízis

Számos, a szoftver fejlődéséhez kapcsolódó szoftverfejlesztési tevékenység során csak a rendszer bizonyos részeit kell egyszerre vizsgálni, és ez az érdekes rész bővül illetve változik a fejlesztési folyamat előrehaladása során. Vagyis, az inkrementális változásokon alapuló szoftver életciklusban [20] a rendszerben eszközölt változás hatásait kell meghatározni, amit aztán változás-kiterjesztésre, regressziós tesztelésre és egyebekre lehet felhasználni. Ezen tevékenységek kulcsa az elemek szomszédainak meghatározása.

Az ilyen „szomszédság” meghatározása jelentősen eltérő lehet attól függően, hogy milyen területen alkalmazzuk. Változás kiterjesztéshez például egy nagyon egyszerű technika az, amikor egy iterációs lépésben egy osztálynak csak a tőle direkt módon függő szomszéd osztályait (az osztály-diagram szerű kapcsolatok alapján) vizsgáljuk meg. Hasonlóképpen, regressziós tesztelésnél egy egyszerű, mégis hatékony technika a tesztelési tűzfal [26, 27], ami csak azon tesztesetek újrafuttatását jelenti, amik a megváltozott rész közvetlen (vagy közeli) függőségeit hajtják végre.

A hatásanalízis [11] célja a szoftverfejlesztés és -karbantartás különböző tevékenységeinek segítése a változások által érintett elemek meghatározása által. Ez rendszerint programelemek közötti különféle relációk meghatározásán keresztül történik. Különböző megközelítések léteznek a hatásanalízist segítő, magasabb szintű szoftverelemek közötti relációk meghatározására [4]. A legtöbb általános hatásanalízis módszer statikus, mint például Rajlich vagy Ren és társaik munkái [20, 21]. A legegyszerűbb statikus módszerek a hívási gráfot vagy valamilyen más könnyűsúlyú függőséget használnak, ami pontatlan vagy nem biztonságos eredményhez vezet (pl. [28]). Lehet pontos és biztonságos módszereket és eredményeket is

találni (például a statikus programszeletelést), de ezen módszerek számításigénye túl magasnak bizonyult [14, 24] a hatásanalízishez.

Az eljárás szintű hatáshalmazokat számító módszereinket Apiwattanapong és társainak [3] a dinamikus *Execute After* (EA) relációja motiválta. Apiwattanapong és társai egy nagyon egyszerű megközelítést használtak ami nagyjából a következő: a program adott futásai alapján egy f függvény potenciálisan hatással van minden olyan metódusra ami bármely említett futás során valamikor utána lett végrehajtva, ami azt jelenti, hogy minden g függvény, amit az f után hajtunk végre része lesz az f hatáshalmazának. Ez egy biztonságos módszer – vagyis egyetlen egy függőséget sem fog kihagyni –, de pontatlan is. Viszont pusztán a függvényhívási információkra támaszkodva lehetetlennek látszik egy ennél pontosabb, de mégis biztonságos módszer megadása.

I/1. *SEA/SEB* relációk definíciója

A komponensek közötti kapcsolatok egy része explicit, mint például objektumorientált rendszerekben az öröklődés, kompozíció, asszociáció, stb. Ezek a függőségek tipikusan meg is jelennek a kódjában explicit hivatkozás formájában. Az explicit függőségek mellett azonban léteznek másfajta függőségek is; ezeket *rejtett függőségeknek*[1] nevezzük. Yu és Rajlich [28] vizsgált egyébként expliciten nem kapcsolódó komponensek között létező adatfolyamokon keresztül közvetített rejtett függőségeket.

Mi egy alternatív módszert ajánlottunk a programkomponensek közötti explicit és rejtett függőségek meghatározására a *Static Execute After* (SEA) és *Static Execute Before* (SEB) relációk definiálásával. A SEA a statikus párja az Apiwattanapong és társai által bevezetett Execute After relációnak [3]. Azt mondjuk, hogy $(f, g) \in SEA$ akkor és csak akkor, ha g valamely része lefuthat f valamely része után a program valamely végrehajtása esetén. A SEA reláció egy lényeges tulajdonsága, hogy biztonságos de pontatlan.

Formálisan a SEA/SEB relációk három (nem diszjunkt) rész-relációra bonthatók:

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

ahol

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

ahol mind a „hívja”, mind a „visszatér” tranzitíven értendő. Hasonlóképpen definiáltuk a *Static Execute Before* SEB relációt is:

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

A SEA reláció kiszámításához szükség van a program megfelelő ábrázolására. A hagyományos hívási gráf (*Call Graph*) [22]) erre nem alkalmas, ugyanis az eljáráson belüli eljáráshívások sorrendjéről semmilyen információt nem tartalmaz. Másrészt viszont az interprocedurális vezérlési folyam gráf (ICFG – *Interprocedural Control Flow Graph*) [19] túl sok információt tartalmaz, így a használata túlságosan drága. Szükség van tehát egy saját reprezentációra.

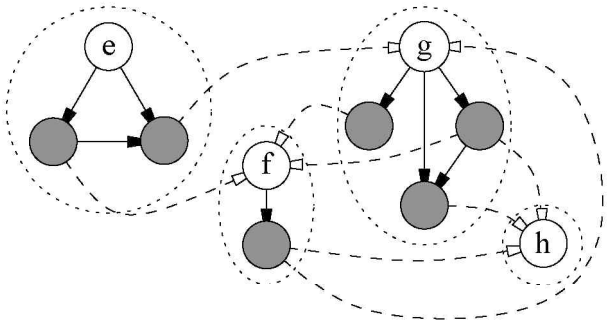
[1] Megjegyezzük, hogy ezek általában csak hatásanalízis előtt rejtettek, egy részletesebb szeletelés ezek nagy részét is képes lenne a megtalálni.

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


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

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

h() {
}
```



1. ábra. Példa: ICCFG

Először definiáltunk egy intraprocedurális komponens vezérlési folyam gráfot (CCFG – *Component Control Flow Graph*), amelyben csak az eljáráshívás szempontjából fontos pontokat és éleket tartjuk számon. Minden egyes CCFG egy eljárást reprezentál, és tartalmaz egy belépési pontot (*entry node*) és számos komponens pontot (*component node*) vezérlési folyam élekkel (*control flow edges*) összekötve. Továbbá, az erősen összefüggő részgráfokat egyetlen pontba nyomjuk össze; azaz ha két hívási hely kölcsönösen elérhető egymásból vezérlési éleken keresztül, akkor hozzájuk ugyanaz a komponens pont tartozik. Az interprocedurális komponens vezérlési folyam gráf (ICCFG – *Interprocedural Component Control Flow Graph*) az egész rendszert reprezentálja oly módon, hogy minden eljáráshoz tartalmaz egy-egy CCFG-t, amiket hívási élekkel (*call edges*) kötünk össze a többi CCFG-vel. Az ICCFG-ben egy c komponens pontból akkor és csak akkor mutat hívási él egy m eljárás belépési pontjára, ha a c pont által reprezentált hívási helyek valamelyike meghívja az m eljárást. Az 1. ábrán egy példa ICCFG látható.

Saját eredmények

A *SEA/SEB* relációk definíciója, illetve a számításukhoz használt *ICCFG* meghatározása szerzőtársaimmal közös eredmény. Az elért eredményeket a [9] cikkben publikáltuk.

I/2. *DFC* metrika definíciója, meghatározása és kiértékelése

A változás kiterjesztésben és regressziós tesztelésben használt hatáshalmaz számító technikák közül a hatékonyság érdekében nagyon sok csak közelítő módszer. Egy lehetőség a pontosság javítására a dinamikus analízis használata a statikus helyett. A dinamikus EA reláció szintén egyszerű és hatékony, de túlságosan konzervatív és így pontatlan. A finomításának ötlete azon a benyomáson alapszik, hogy minél „közelebb" van egy f függvény végrehajtása egy g függvény végrehajtásához a program valamely futása során, annál valószínűbb, hogy függenek egymástól.

Mielőtt megadnánk a formális definíciót, bevezetjük a *dinamikus hívási fa* (*dynamic call tree*) fogalmát. Ez egy gyökeres fa rendezett élekkel, ahol az f függvénnyel címkézett p pont az f függvény egy meghívott példányát jelöli, és egy $p \to q$ él az f-nek a p példányából történő g hívást reprezentál annak q példányába, ahol a q címkéje g. Szintén használni fogjuk az $f \to g$ *hívási lánc* fogalmát, ami egy útvonal p és q pontok között, ahol a p és q rendre f és g függvényekkel címkézett pontok, és amire igaz, hogy a fa gyökerétől p-ig tartó

útvonal prefixe a fa gyökerétől q-ig tartó útvonalnak.

Most megadjuk az *Execute After* reláció egy d indirekciós szint szerinti kiterjesztését. Formálisan:

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

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

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

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

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

és a fenti két reláció kombinálásával megkaphatjuk az *Execute Round* ($ER^{(d)}$) relációt is a következőképpen:

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

Megfigyelhető, hogy a mi definíciónk speciális esete, az $EA^{(\infty)}$ megfelel az Apiwattanapong és társai által megadott *Execute After* reláció definíciójának, míg $ER^{(\infty)}$ a végrehajtott metódusok közötti teljes relációt (teljes gráfot) jelöli.

Természetesen, ha egy d érték elegendő két metódus *Execute Round* relációval való összekapcsolásához, akkor ezt a két függvényt az ER minden ennél magasabb d' érték esetén is összekapcsolja. A *Dynamic Function Coupling* (DFC) metrika tehát minden f, g metódus-párra azt a legalacsonyabb d értéket határozza meg, amelyre a két metódus $ER^{(d)}$ relációban van:

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

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

A fentiek alapján egy rögzített d értékre, a változott metódusok C halmazával számolt dinamikus változás hatáshalmaz a következő:

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

Algoritmusok

Bemutattunk három, belépési (*function entry*) és visszatérési (*function return*) eseményeket tartalmazó végrehajtási nyomon dolgozó algoritmust.

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

Az első egy globális rekurzív algoritmus, ami a DFC értékét számolja ki minden függvénypárra legrosszabb esetben $O(t \cdot n^2)$ idő- és $O(n \cdot m)$ tárigénnyel, ahol n a függvények száma, m a hívási fa magassága, t pedig a végrehajtási nyom hossza.

A második egy igényvezérelt algoritmus a hatáshalmaz számításra egy paraméterként kapott d indirekciós értékkel. Ennek a legrosszabb esetben $O(t \cdot n)$ a számítási és $O(n \cdot m)$ a tárhely komplexitása.

A harmadik algoritmus is egy igényvezérelt algoritmus a hatáshalmaz számításra de rögzített $d = 1$ indirekciós értékkel. Ennek $O(t \cdot n)$ az idő- és $O(n \cdot m)$ a tárigénye a legrosszabb esetben. Ez olyannak látszik, mint az előző algoritmusé. De míg az átlagos esetben nem sokat javul a legrosszabb esethez képest, addig ennek az idő- és tárigénye majdnem $O(t)$ és $O(n + m)$-re csökken.

Mérések

Elvégeztünk néhány kísérletet. Három nyílt forráskódú Java programon mértük a relációk és rész-relációk pontosságát és teljességét. Ezeket az értékeket a programszeletelés eredményét pontosnak tekintve mértük. A megállapításaink összegzésére, megadjuk a mérések előtt felvetett kérdésekre a válaszokat:

1. *Igaz-e, hogy két függvény közötti alacsony DFC érték nagyobb valószínűséggel jelent kapcsolatot kettejük között?* Igen. Leginkább az 1-es és 2-es DFC értékek jelentenek nagyobb valószínűséget, mint a magasabb indirekciós szintek.

2. *A hívási rész-reláció önmagában és a szekvenciális rész-relációval együtt milyen mértékben tükrözi a valós csatolásokat?* Magas indirekciós szintnél a hívási rész-reláció nem sokat tartalmaz a valódi csatolásokból (csak körülbelül 20%-it talál meg). Ez azt jelenti, hogy a csatolások javát a szekvenciális rész-reláció fedi le, tehát az egyszerű csak hívás alapú algoritmusok nem kielégítőek.

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

4. *Milyen d értéket használjunk, ha a pontosság a fontos, és ilyenkor milyen a teljesség?* A legjobb pontosságot $d = 1$ vagy $d = 2$ esetén kapjuk. A teljesség viszont ekkor még nagyon alacsony, 20% körül mozog.

5. *Az eredeti EA relációhoz képest mekkora hatáshalmaz méret csökkenéssel számolhatunk?* A legalacsonyabb 1-es érték esetén a hatáshalmazok mérete 13–15%-a a biztonságos módszerének, míg 2-es szinten ez 25–35%.

Saját eredmények

A *DCF* metrika definíciója és kiszámításának elve, valamint a metrika mérésekkel történő kiértékelése szerzőtársaimmal közös eredmény. A *DFC* metrikát és a metrikán alapuló hatáshalmazokat számító algoritmusok kidolgozása saját eredmény. Az elért eredményeket a [6] cikkben publikáltuk.

II. Alacsony szintű analízis

A programszeletelés egyszerre hasonlít és különbözik a hatásanalízistől. Hasonlít, mert mindkettőnek ugyanaz a célja: programelemek közötti relációk felderítése. Mégis különbözik, hiszen alacsony szintű és pontos relációkat ad, de sokkal nagyobb számításigénnyel.

Idővel sok programszeletelési módszerek lett kidolgozva [24, 25]. A módszerek jelentős része programelemek (változók, utasítások, címek, predikátumok, stb.) közötti különféle (adat és vezérlési) *függőségek* alapján számol szeleteket. Részletes statikus szeletelő módszerekből sok publikáció született. Például Horwitz és társai [14] rendszerfüggőségi gráf (*System Dependence Graph* – SDG) alapú munkája szolgált számos további implementáció és javítás kiindulópontjaként.

Az alapvető dinamikus szeletelő módszerek többféle elvre épülnek, mint például Korel és Laski [17, 18], Agrawal és társai [1, 2] illetve Kamkar és társai [16] módszerei. Az Agrawal és Horgan [2] által publikált tradicionális dinamikus függőség alapú módszer *dinamikus függőségi gráfokat* (*Dynamic Dependence Graph* – DDG) használ, ami minden egyes utasítás minden egyes előfordulásához (minden *akcióhoz*) egy külön csúcsot rendel, az élek pedig a program egy konkrét futása során keletkező dinamikus függőségeket reprezentálják. A dinamikus szelet kiszámítása ezen a gráfon a kritérium csúcsából elérhető összes csúcs megtalálását jelenti.

Meglepően kevés publikáció jelent meg viszont, ami a dinamikus szeletelés gyakorlati oldalával foglalkozna és részletes algoritmusokat adna. Ennek egyik oka lehet, hogy programok dinamikus analízise számos okból kifolyólag egy eredendően nehéz probléma, amik közül az egyik legjelentősebb a program futása során keletkező nagyszámú esemény. Az alap dinamikus szeletelő algoritmusok legtöbbjének problémát okoz a nagy input kezelése. A DDG méretét például a végrehajtási nyom elemeinek száma határozza meg, ami korlátlan.

Gyimóthy Tibor, Forgács Gábor és Beszédes Árpád elkészített egy olyan hátramutató szeleteket számító dinamikus algoritmust, amely a végrehajtási történet egyszeri végigjárásával az összes lehetséges dinamikus kritériumra kiszámolja a megfelelő szeleteket [13]. Ez alapján Beszédes Árpád kidolgozott egy algoritmust egyetlen szelet kiszámítására [5]. Az algoritmusok az egyes utasítások reprezentálására úgynevezett $d : U$ (*definíció-használat*) párokat használnak. Bár az eredeti algoritmusok csak egy nagyon egyszerű programozási nyelvet támogatnak, viszonylag alacsony tárigényük miatt alkalmasak nagyméretű programok szeletelésére is.

II/1. *d:U* alapú szeletelő algoritmusok meghatározása

A Beszédes Árpád és szerzőtársai által kidolgozott két algoritmus alapján nyilvánvalóvá vált, hogy ugyanezzel a reprezentációval többféle gráfmentes dinamikus szeletelő algoritmus is megalkotható. Ezért azután meghatároztuk a fentebb említett algoritmusok néhány jellemzőjét, és meghatároztuk a lehetséges értékeiket, azután megvizsgáltuk ezek kombinációit. Három jellemzőt találtunk:

Szeletek iránya. A két alapvető szeletelési irány az *előremutató* és a *hátramutató* szeletelés. Az előremutató szeletelés esetén azokra a programpontokra vagyunk kíváncsiak, amelyek a szeletelési kritérium által meghatározott ponton kiszámított értékeket a program futása során később (akár áttételesen is) felhasználják. Egy hátramutató szelet azokból az utasításokból áll, amik hatással lehetnek egy adott programponton kiszámított értékre.

Globális vagy igényvezérelt. A tradicionális megközelítés szerint egyszerre egy kritériumunk van, és az ehhez tartozó egyetlen szeletet számítjuk ki. Ezt nevezzük *igényvezérelt* szeletelésnek. Lehetőség van viszont arra, hogy a végrehajtási történet egyszeri

feldolgozásával egyszerre több (akár az összes lehetséges) szeletet kiszámoljunk. Ebben az esetben *globális* szeletelésről beszélünk.

Feldolgozás iránya. A végrehajtási nyomot szintén kétféleképpen dolgozhatjuk fel. Az *előrefelé haladó* feldolgozás a „természetes" irány, hiszen a végrehajtási történet ilyen módon keletkezik. Néha csak ez a fajta feldolgozási irány valósítható meg. Ugyanakkor előfordulnak olyan szituációk, amikor a *hátrafelé haladó* feldolgozás is alkalmazható és hatékonyabb, mint a másik irány.

Ez összesen nyolc lehetőség, ami közül néhány hasznos algoritmust ad, ugyanakkor vannak értelmetlen kombinációk is. A lehetőségeket az 1. táblázatban foglaltuk össze.

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

1. táblázat. Dinamikus szeletelő algoritmusok áttekintése

Dinamikus szeleteket igényvezérelt módon számolni azt jelenti, hogy a dinamikus szeletet a program egy adott futása és egy adott kritériumra számoljuk. Bejárjuk a végrehajtási utat a kritériumban meghatározott akciótól kezdve, és a $d : U$ reprezentáció segítségével követjük a dinamikus függőségeket a szeletelés irányától függően hátrafelé az első végrehajtott utasítás irányába vagy előre a nyom vége felé. Így összesen két igényvezérelt dinamikus szeletelő algoritmust kapunk.

A szeleteket igényvezérelt módon ellentétes szeletelési és feldolgozási iránnyal számítani értelmetlen. Ez gyakorlatilag egy globális algoritmust eredményezne, mert az összes érintett akcióhoz tartozó szeletet meg kellene tartani amíg a végrehajtási nyomban el nem érjük a kritérium akcióját.

Számos alkalmazásban a program egy adott futásának több szeletére is szükségünk lehet egyszerre. Ez vezet az ötlethez, hogy a végrehajtási nyom egyszeri feldolgozásával egyszerre több dinamikus szeletet is kiszámoljunk. A sok szelet kiszámítása lehetséges igényvezérelt algoritmusok párhuzamos futtatásával: előremutató szeleteket előrehaladó feldolgozással, hátramutató szeleteket visszafelé haladó feldolgozással számolva. Ugyanakkor ez a megközelítés nem túl praktikus, mivel az adatstruktúrákat (és a szeleteket) az végrehajtási nyom teljes feldolgozása alatt karban kell tartani minden dinamikus kritériumhoz.

Szerencsére lehetséges sokkal praktikusabb globális algoritmusok megalkotása is, amikben nem kell a teljes szeleteket tárolni a végrehajtási nyom feldolgozása alatt, elég csak a program változóihoz tartozó függőségi halmazokat. Ezek a függőségi halmazok utasítás sorszámokat tárolnak, megadva ezzel az adott pont adott változóinak aktuális függőségeit. Ezeket a halmazokat a $d : U$ információkból számítjuk, és minden végrehajtási lépésben frissítjük. Így pusztán ezen halmazok aktuális értékei segítségével képesek vagyunk az összes kritériumhoz kiszámolni a dinamikus szeleteket. Érdekes tulajdonsága ennek a megközelítésnek, hogy az említett halmazok a nyom fordított (szeletelési iránnyal ellentétes) irányú feldolgozásával kaphatóak meg.

DDG ekvivalencia

Ahhoz, hogy megmutassuk, hogy az általunk használt $d : U$ alapú algoritmus ugyanazokat a szeleteket számolja mint a DDG alapú algoritmus, először a két reprezentáció ekvivalenciáját kell megmutatnunk. Legyenek a program utasításai az $i \in \{1, \ldots, I\}$ értékekkel azonosítva. Adott ugyanannak a programnak a PDG (*Procedure Dependency Graph*, az SDG egy alkotóeleme) és $d : U$ reprezentációja. Definíció szerint:

$$\begin{array}{rcl} \text{in } \mathbf{PDG} & & \text{in } d : U \\ \exists P_i & \Longleftrightarrow & \exists d_i : U_i \\ \exists P_i \rightarrow P_k \text{ vezérlési függőségi él} & \Longleftrightarrow & d_i \text{ predikátum változó} \in U_k \\ \exists P_i \rightarrow P_k \text{ adatfüggőségi él} & \Longrightarrow & d_i \in U_k \end{array}$$

Ezek alapján viszont megmutatható, hogy akkor és csak akkor létezik $P_{i^j} \rightarrow P_{k^l}$ él a DDG gráfban, ha $d_i \in U_k$ és a d_i változót az l lépés előtt utoljára definiáló akció i^j ($LD(d_i, l) = i^j$).

Igényvezérelt algoritmusok esetén az eredmény ekvivalenciáját úgy mutattuk meg, hogy a leírt algoritmust ekvivalens átalakításokkal egy gráfszínezési algoritmussá alakítottuk. Ez egy adott kiindulópontból az élek mentén elérhető gráfpontokat színezte be. A színezett pontokhoz tartozó utasítások pont a DDG-ből előállított szeletet alkotják.

Annak bizonyítására, hogy a globális algoritmusok is a DDG alapú szeleteléssel kapott szeleteket számolják ki teljes indukciót alkalmaztunk. Kezdetben az algoritmusok által használt halmazok üresek, ami az első akció feldolgozása előtt triviálisan megfelel az üres szeleteknek. Majd feltettük, hogy az i^j akciót feldolgozó iteráció kezdetén a halmazok a DDG-ben a megfelelő helyekről elérhető utasításokat vagy akciókat tartalmazzák. Végül az algoritmusokról megmutattuk, hogy ha a feltevéseink igazak az i^j feldolgozása előtt, akkor utána is igazak lesznek.

Saját eredmények

Az algoritmusok osztályozása, négy új algoritmus (igényvezérelt előremutató előrehaladó feldolgozással, globális előremutató előrehaladó és fordított feldolgozással, globális visszamutató fordított feldolgozással) meghatározása és kidolgozása szerzőtársaimmal közös eredmény. Az algoritmusok *DDG* algoritmussal való ekvivalenciájának igazolásában a saját hozzájárulásom a meghatározó. Az eredményeket a [7] cikkben és az ezt kiegészítő [8] jelentésben publikáltuk.

II/2. *d:U* alapú szeletelő algoritmusok implementációja

A dinamikus szeletelő algoritmusainkat C és Java nyelvekre implementáltuk. Valódi C programok szeletelése során számos problémát kellett megoldani, mint például a *mutatók*, *függvényhívások* és *ugró utasítások* kezelése. Elsőként átalakítottuk a $d : U$ reprezentációt a C nyelv igényeinek megfelelően. C programok esetén a $d : U$ reprezentáció $d : U$ elemek sorozatát tartalmazza:

$$i.\ \langle (d_1 : U_1), (d_2 : U_2), \ldots \rangle \ .$$

A sorozat elemeinek sorrendje fontos, ez pedig a részkifejezések végrehajtási sorrendje.

A végrehajtási utat szintén módosítottuk. Hozzáadtunk pár technikai adatot, mint például a memóriacímeket, blokk belépési/kilépési eseményeket, függvény hívási/visszatérési eseményeket, stb. Ezt a kiterjesztett *EH*-t *TRACE*-nek hívjuk. A *TRACE*-t a program instrumentálásával (utasítások beszúrása), majd az instrumentált verzió futtatásával kapjuk.

A mutatók kezelését minden változó (amelyiknél ez lehetséges) memóriacímre konvertálásával oldottuk meg. Így az algoritmus futása során egy másik, úgynevezett *dinamikus* $d : U$ felépítésére lesz szükség. Ez a *dinamikus* $d : U$ tartalmazza a memóriacímeket, amiken az algoritmus dolgozik.

Mutatók kezelése

Egy változó címe a hatókörén belül nem változik, így miután meghatároztuk, akárhányszor használható. De egy mutató értékét minden egyes felhasználás alkalmával le kell kérdeznünk, hiszen bármikor megváltozhat. Ezért az instrumentált program kiírja ezeket a címeket a *TRACE*-be a `remember()` (változókhoz) és `dump()` (mutatókhoz) függvények segítségével:

	`int x, *p;`	`int x, *p;`
		`remember("x", &x, sizeof(int));`
		`remember("p", &p, sizeof(int*));`
1.	`x=1;`	`x=1;`
2.	`p=&x;`	`p=&x;`
3.	`*p=2;`	`*dump("PTR1", p,, sizeof(int))=2;`
4.	`print(x);`	`print(x);`

A program statikus és dinamikusan feloldott $d : U$ reprezentációja és a 4. sorhoz számított szelet – feltéve, hogy az x és p változók címei `01` és `02` – a következő:

line	def	:	USE	akció	def	:	USE	$Szelet$
1	x	:	$\emptyset$	1^1	`01`	:	$\emptyset$	$\emptyset$
2	p	:	$\emptyset$	2^2	`02`	:	$\emptyset$	$\emptyset$
3	$PTR1$	:	$\{p\}$	3^3	`01`	:	{`02`}	$\{2\}$
4	OUT	:	$\{x\}$	4^4	OUT	:	{`01`}	$\{2,3\}$

A C nyelvben a tömbök és a mutatók gyakorlatilag ugyanazok, és a konverzió egyikről a másikra meglehetősen egyszerű. Egy `t` tömb i-ik elemét `t[i]`-ként írjuk, de kifejezhető `*(t+i)` alakban mutatóként is. Így ha egy tömb valamelyik elemére hivatkozunk, azt a $d : U$-ban mutatóként kezeljük, és így kiírjuk a címét.

A struktúrák mezőinek struktúrán belüli címe (offszet) statikus időben meghatározható, de ebből az információból dinamikus címeket előállítani meglehetősen komplikált lenne. Ehelyett a struktúrák mezőit is inkább eleve mutatókként kezeljük. Ezáltal a struktúramező elérést is visszavezettük a mutatók kezelésére. A struktúrákat magukat nem konvertáljuk, azokat általános változóként kezeljük:

De a memóriacím önmagában nem írja le pontosan a változót. Egy struktúra és első mezőjének címe például ugyanaz, de egy új érték hozzárendelése a teljes struktúrához a mezőin keresztüli függőségeket indukál. Ezért a `remember()` és `dump()` függvények a memóriaméretet is rögzítik.

Algoritmus

A C programokat szeletelő módszerünk a következőképpen működik. Először elemezzük és instrumentáljuk a programot, és előállítjuk a statikus $d : U$ reprezentációját. Azután az instrumentált programot lefordítjuk és futtatjuk, aminek eredménye a *TRACE*. Végül a dinamikus szeletelési algoritmust futtatjuk az előzőleg felépített $d : U$ reprezentáció és a *TRACE* segítségével.

A *TRACE* feldolgozása és a változók memóriacímekké konvertálása érdekében az algoritmusok *TRACE* kezelő ciklusa a következőképpen módosul. A *TRACE* aktuális elemének típusa alapján a következő tevékenységeket kell elvégezni.

- *függvény belépési esemény*: Az aktuális $d : U$ elem feldolgozását felfüggesztjük, a pozíciót pedig egy verembe mentjük.

- *függvény kilépési esemény*: A feldolgozás a verem tetején elmentett $d : U$ pozícióban folytatódik. A pozíciót kivesszük a veremből.

- *EH elem*: Az aktuális akció az *EH* elem által meghatározott akció, a végrehajtás ennek első $d : U$ elemével folytatódik.

- *egyéb*: Ez alapján a feloldatlan hivatkozásokat fordítsuk le memóriacímre.

A statikus $d : U$ változókat a típusuk alapján oldjuk fel a dinamikus $d : U$-ban:

- *Skaláris változók.* A deklarációtól kezdve a hatókörükön belül állandó címük van. A címük a C program veremműködésének szimulálásával (címeket és blokk belépési/kilépési eseményeket használva) feloldható. A dinamikus $d : U$ az így kapott címet használja.

- *Dereferencia változók.* Jelölésük $PTRn$, ahol n egy globális számláló az összes dereferenciához. A $TRACE$-ben található információk alapján közvetlenül feloldható.

- *Predikátum változók.* Az ilyen változókat Pn-nel jelöljük, ahol n a megfelelő predikátum utasítás sorszáma. A dinamikus $d : U$-ban a program függvényhívási vermének mélységével kiegészítjük, elkerülendő a rekurzív hívásból eredő ütközéseket.

- *Kimeneti változók.* A jelölése $OUTn$, ahol n szintén utasítás-sorszámot jelöl. Definíció szerint a kimeneti változó egy olyan ál-változó, amit azokra a helyekre generálunk, ahol az U halmazt használjuk ugyan, de semmilyen más változó nem kap értéket U-ból. A dinamikus $d : U$-ban változatlanok maradnak.

- *Függvényhívás argumentum változó.* Jelölésük $ARG(f, n)$, ahol f a függvény neve n pedig az argumentum sorszáma. Egy argumentum változót a függvényhívás helyén *definiálunk* és a függvény belépési pontján *használunk*. A dinamikus $d : U$-ban változatlanok maradnak.

- *Függvényhívás visszatérési változó.* Ezeket $RET(f)$-fel jelöljük, ahol f a függvény neve. Egy ilyen változót a függvény kilépési pontján *definiálunk* és a függvény hívásának helyén *használunk*. A dinamikus $d : U$-ban változatlanok maradnak.

Ezután, ha az aktuális $d : U$ elem feldolgozható (pl. nincs benne fel nem oldott változó), akkor feldolgozzuk.

Saját eredmények

Az algoritmusok C nyelvű programokat szeletelő változatának kidolgozása és implementációja során a változók kezelését, azaz a forráskódban látható hivatkozások és a futás közbeni memóriacímek összerendelésének problémáját oldottam meg. Az eredményeket a [10] és [12] cikkekben publikáltuk.

II/3. *d:U* alapú szeletelő algoritmusok kiértékelése

Kétféle kiértékelést végeztünk. Először, elemeztük a hat szeletelő algoritmusunk komplexitását, és összehasonlítottuk a DDG alapú algoritmuséval. Másodszor, különféle méréseket végeztünk a C és Java implementációkkal.

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

2. táblázat. Dinamikus szeletelő algoritmusok számítási komplexitása

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

3. táblázat. Dinamikus szeletelő algoritmusok tárigény komplexitása

Komplexitás

Amikor az idő és tárigényt taglaljuk, kifejezetten az algoritmusok lényegi részére koncentrálunk. Nem értjük bele például a végrehajtási nyom beolvasását és tárolását, vagy program statikus reprezentációjának felépítését és tárolását. Kihagyjuk a konkrét megvalósításokhoz szükséges módosításokat is.

A hat $d : U$ alapú és a DDG alapú algoritmus legrosszabb és átlagos idő- és tárigényére vonatkozó számításainkat a 2. és a 3. táblázatokban foglaltuk össze. A használt jelölések: a végrehajtási történet hossza J; a program utasításainak darabszáma I; a program változójának száma V; a futás során valóban definiált változók száma V^{DEF}; DS az átlagos szeletméret; DEP a DDG egy pontjából egy adott irányban elérhető pontok átlagos száma. (DS az I-vel áll kapcsolatban, míg DEP pedig a J-vel.) Az igényvezérelt algoritmusoknál feltüntetett értékek egyetlen szelet kiszámítására vonatkoznak, míg a praktikus és párhuzamos algoritmusoknál az összes lehetséges szeletre.

Átlagos esetben az igényvezérelt algoritmusaink hatékonyabban lehetnek a DDG algoritmusnál, mert egyszeri bejárással meg is határozzák a szeletet, ehhez nem kell külön lépés, mint a DDG-nél. Ráadásul egyszerre mindig csak korlátozott mennyiségű dinamikus függőséget tárolnak a memóriában, így tárigényük is alacsonyabb (nem csak az előremutató algoritmusnál, ahol ez $V^{DEF} \leq J$ miatt nyilvánvaló).

A praktikus algoritmusaink átlagos időigénye nem kifejezetten jobb vagy rosszabb, mint a DDG algoritmusé. Mivel DS a program méretével arányos így korlátos, DEP viszont az EH méretével arányos tehát potenciálisan korlátlan, megfelelően nagy végrehajtási nyom esetén a $d : U$ alapú algoritmus tűnik használhatóbbnak. Az algoritmusaink $O(V^{DEF} \cdot DS)$ átlagos tárigénye (figyelembe véve, hogy valós alkalmazásokban a V^{DEF} értéke inkább V, mint J függvénye) praktikusabb mint a DDG $O(J)$ tárigénye.

A párhuzamos algoritmusaink viszont mind idő- mind tárigény tekintetében nyilvánvalóan rosszabbak mint a DDG alapú algoritmus.

Mérések a C implementációval

A C nyelvű implementációval végzett mérések célja elsősorban az algoritmusok gyakorlati használhatóságának igazolása volt. A praktikus és igényvezérelt hátramutató szeleteket számító algoritmus C nyelvű implementációját öt kisebb programon mértük: *bcdd*, *unzoo*, *bzip*, *bc* és *less*. A mérések során a tesztprogramok különböző tulajdonságait, és az algoritmusok komplexitásban megjelenő jellemzőit rögzítettük.

Megállapításaink a következők voltak:

- A mérések során nem találtunk kapcsolatot a szeletméret és a végrehajtási nyom hossza között.
- A statikus (programkódban jelenlévő) és dinamikus (futás közben foglalt) változók számának korrelációja viszonylag magas, 0.73. Az eredmények alapján a program változóinak memóriacímre cserélése nem okoz a változók számának megtöbbszöröződéséből adódó problémát.
- A gyakorlati használhatóság miatt fontos, hogy a halmazok mérete és a halmazműveletek száma vajon milyen viszonyban áll a program illetve a végrehajtási nyom méretével. A maximális halmazméretek és a lépésenkénti halmazműveletek átlagos száma többékevésbé a programmérettel együtt változott. Megállapítottuk továbbá, hogy az egyes futások során a halmazok maximális mérete nem nőtt szignifikánsan a nyom feldolgozásának előrehaladtával.
- Az igényvezérelt algoritmus esetén a hosszabb végrehajtási nyomból nem következett az algoritmus iterációs számának növekedése.
- Az igényvezérelt algoritmus fő ciklusának iterációs számát befolyásoló halmaz mérete nagymértékben a szeletek méretével korrelált.

Összességében tehát elmondhatjuk, hogy az algoritmusok futásidejét meghatározó tényezőknek (dinamikus változók száma, halmazműveletek száma) leginkább statikus összetevőik vannak, és az igényvezérelt hátramutató algoritmus lépésszáma jóval az *EH* mérete alatt van.

Mérések a Java implementációval

A Java implementációval végzett méréseink leginkább a különféleképpen előállított szeletek méretére vonatkoztak, nevezetesen a statikus, dinamikus és uniós szeletek viszonyára. A statikus szeletek előállításához az *Indus* [15] nevű Java statikus szeletelő eszközt használtuk.

A mérésekhez összesen öt kis Java programot használtunk (*RayTracer*, *JSubtitles*, *NanoXML/DumpXML*, *java2html* és *dynjava*) körülbelül 100 tesztesettel programonként. A futáshosszak statisztikái a a 4. táblázatban láthatók.

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

4. táblázat. Végrehajtott utasítások

Megállapításaink a következők voltak:

- Az uniós szeletek jóval kisebbek, mint a statikus társaik.
- Az előremutató uniós szeletek mérete kisebb, mint a hátramutató szeleteké.
- Sokkal több kis szelet található az előremutató szeletek között, de a maximális méretek nagyjából megegyeznek a hátramutató szeletek maximális értékeivel.
- Az uniós szelet előállítása során a szeletméretek és a lefedettség között számított korreláció értéke 0.89 és 0.96 között mozgott. Ez jó, mert a lefedettség mértéke pontosan mérhető, és ezzel a végleges szeletméret becsülhető.

Fontosabb eredmény tehát, hogy az uniós szelet mérete jóval a statikus szelet mérete alatt van, illetve, hogy a növekedése (az újabb dinamikus szeletek hozzáadásával) erőteljesen korrelál az utasítás lefedettség növekedéssel.

Saját eredmények

Az elméleti algoritmusok elvi kiértékelése saját eredmény, melyet összefoglaló jelleggel a [7] cikkben, részletesebben a [8] jelentésben publikáltuk. A C nyelvű implementáció kiértékelése közös eredmény, amit a [10], [12] cikkekben és [8] jelentésben publikáltunk. A Java nyelvű implementációval történt mérések kiértékelése szintén közös eredmény, mely az [23] cikkben lett publikálva.

Köszönetnyilvánítás

Az értekezés olyan eredményeket foglal össze, melyek teljes megvalósítása csapatmunka eredménye volt. Néhol ugyan világos határok húzódtak az egyes résztvevők munkái között, de az eredmények külön-külön nehezen értelmezhetők. Köszönet témavezetőmnek Gyimóthy Tibornak az útmutatásért, szerzőtársaimnak Beszédes Árpádnak, Faragó Csabának, Tóth Gabriellának, Jász Juditnak, Fischer Ferencnek, Szabó Zsoltnak, Szegedi Attilának, Faragó Szabolcsnak, Václav Rajlichnak, Csirik Jánosnak, és munkatársaimnak Havasi Ferencnek, Kiss Ákosnak, Vidács Lászlónak, Siket Istvánnak, Ferenc Rudolfnak és azoknak, akiket név szerint nem említettem a közös munkáért.

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

Hivatkozások

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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