# C Programok Dinamikus Szeletelése

*Gergely Tamás*, `gertom@inf.u-szeged.hu`,
*Faragó Csaba*, `h633185@stud.u-szeged.hu`

Témavezető: *Dr. Gyimóthy Tibor*

2019. február 20.

## 1. BEVEZETÉS

A programszeletelési módszerek széles körben alkalmazhatóak hibakeresésre (*debugging*), tesztelésre és karbantartásra ([7], [18], [5], [8]). Egy szelet tartalmazza az összes olyan utasítást és predikátumot, amely hatással lehet változók egy $V$ halmazára a program adott $p$ pontján ([21]). Egy szelet lehet egy futtatható program vagy a programkód egy részhalmaza. Az első esetben a redukált program viselkedése egy $v$ változó szempontjából a program $p$ pontján ugyanaz, mint az eredeti programé. A második esetben a szelet azon utasítások halmaza, melyek befolyásolhatják a $v$ változó $p$ pontbeli értékét. A szeletelő algoritmusok osztályozhatók aszerint, hogy csak statikus információkat használnak (*statikus szeletelés*), vagy egy adott inputra számolják ki, hogy mely utasítások befolyásolják egy változó értékét (*dinamikus szeletelés*).

Sok alkalmazásban (pl. hibakeresés) a dinamikus szeletelés sokkal előnyösebb, hiszen pontosabb eredmének előállítására képes (vagyis a dinamikus szelet kisebb, mint a statikus).

Többféle különböző szeletelési módszer létezik ([13], [12], [3]). Agrawal és Horgan egy pontos szeletelési módszert írt le ([3]), amiben a dinamikus függőségeket gráffal reprezentálták. Ez a Dinamikus Függőségi Gráf (DDG – Dynamic Dependence Graph) egy utasítás minden előfordulásához tartalmaz egy külön csúcsot. A DDG alapján egy $v$ változóhoz számolt dinamikus szelet azokat az utasításokat tartalmazza, amelyeknek hatása van $v$ értékére. Ennek a megközelítésnek a legnagyobb hátránya az, hogy a DDG mérete a végrehajtott utasítások számával arányos. Bár Agrawal és Horgan ajánlott egy módszert a DDG csökkentésére, még ez a redukált DDG is nagyon nagy lehet. Emiatt ez az eljárás nem alkalmazható valódi méretű programokra, ahol adott esetben több millió lépés is végrehajtódhat.

Gyimóthy Tibor és társai kidolgoztak egy módszert a dinamikus szeletek számolására ([9]). Az általuk adott eljárás a program elejétől kezdve folyamatosan számolja a szeleteket, így egy adott lépésben az összes változó aktuális

szeletét megadja. Az eljárást azonban ,,csak" egy egyszerű programnyelvre dolgozták ki. Ebben a dolgozatban ezt a módszert fejlesztettük tovább úgy, hogy alkalmas legyen C nyelvű programok szeletelésére is. A bővített algoritmus kezeli a C nyelv kifejezéseit, a mutatókat, a függvényeket és az ugró utasításokat is. Ennek a megközelítésnek a legnagyobb előnye, hogy valódi méretű C programokra is alkalmazható, hiszen a memóriaigénye a program által használt különböző memóriahelyek számával, és nem a végrehajtott lépések számával arányos. Tapasztalataink szerint a különböző memóriahelyek száma sokkal kevesebb, mint a végrehajtott lépések száma.

A dolgozat témájából tudományos dolgozat is készült, ami első helyen került elfogadásra a 2001. március 14-16 között Portugáliában megrendezendő **CSMR 2001** (5th European Conference on Software Maintenance and Reengineering) nemzetközi konferencián. A kidolgozott módszert a Szegedi Tudományegyetem Mesterséges Intelligencia Kutatócsoportján egy projektben implementáljuk.

A dolgozat az alábbi módon épül fel. A következő fejezetben a dinamikus szeletelés alapfogalmait ismertetjük. A 3. fejezetben a Gyimóthy Tibor és társai által kidolgozott módszert mutatjuk be. A 4. fejezetben részletesen leírjuk az előző, 3. fejezetben bemutatott algoritmus kibővítését.

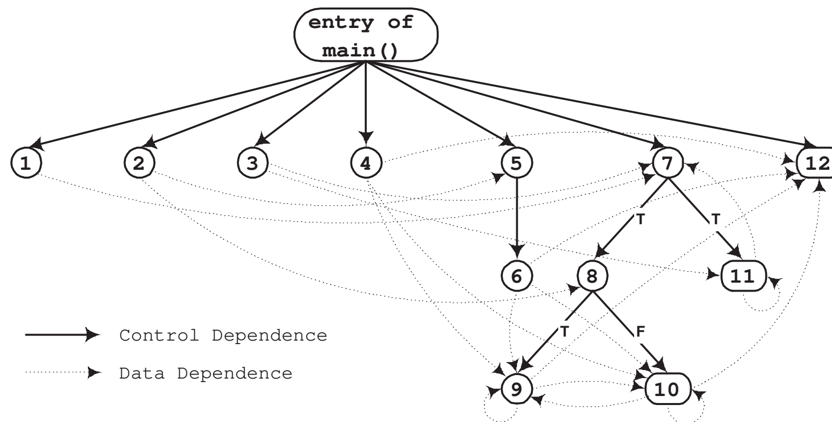
# 2. DINAMIKUS SZELETELÉS

Néhány esetben a statikus szeletek felesleges utasításokat is tartalmaznak. Ilyen eset például a hibakeresés, amikor dinamikus információk is a rendelkezésünkre állnak. A dinamikus szeletelés célja az volt, hogy sokkal pontosabban lehessen meghatározni azokat az utasításokat, melyek a hibát tartalmazhatják, feltéve, hogy a hiba egy adott bemenetre fordult elő.

Először röviden bemutatjuk a *programfüggőségi gráfot* (PDG – Program Dependence Graph; [17], [10]). Ez a gráf programok reprezentálására alkalmas. A PDG-ben a program minden utasításának és predikátumának megfelel egy csúcs. Ezeket a csúcsokat kétféle irányított él kötheti össze: *adatfüggőségi él* (data dependence edge) vagy *kontrolfüggőségi él* (control dependence edge). A PDG-ben akkor van $n$ csúcsból $m$ csúcsba mutató adatfüggőségi él, ha az $m$ csúcshoz rendelt utasításban használjuk az $x$ változó értékét, amit az $n$ csúcshoz rendelt utasításban definiáltunk, és a program *vezérlési folyamat gráfjában* (CFG – Control Flow Graph) van olyan út $n$-ből $m$-be, amely út mentén sehol sem definiáljuk $x$-et. A $p$ csúcsból akkor mutat kontrolfüggőségi él az $n$ csúcsba, ha (1) a CFG-ben a $p$ csúcsból két él indul ki, és ha (2) az egyik élen elindulva biztosan áthaladunk az $n$ csúcson, míg a másik élen elindulva lehetséges, hogy elkerüljük azt. (Azt mondhatjuk, hogy az $n$ csúcshoz rendelt utasítás végrehajtása közvetlenül függ a $p$ csúcshoz rendelt predikátum kiértékelésétől.) Az 1. ábrán bemutatunk egy rövid példaprogramot és a program PDG-jét.

A PDG alapján egy program statikus szeletelése könnyen elvégezhető. Az $n$ csúcsban előforduló $v$ változóhoz tartozó szelet azokat a csúcsokat tartalmazza, melyek hatással lehetnek a $v$ változó $n$ csúcsbeli értékére. A példánkban a 12-es sorban szereplő $s$ változóhoz tartozó szelet a program összes utasítását

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




1. ábra. A példaprogram és a PDG-je

tartalmazza.

A különböző dinamikus szeletelési módszerek megismeréséhez néhány fogalom ismerete szükséges. Ezeket az 1. ábrán található példán keresztül magyarázzuk el.

A végrehajtott utasítások sorozatát *végrehajtási út*-nak (execution history) nevezzük és $EH$-val jelöljük. Legyen a példánk bemenete $a = 0, n = 2$. Az $EH$ ekkor a következő: $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 \rangle$. Láthatjuk, hogy az

$EH$ olyan sorrendben tartalmazza az utasí tásokat, amilyen sorrendben azok végrehajtódtak. Így $EH(j)$ megadja a $j$-edik lépésben végrehajtott utasítás sorszámát.

Egy utasítás két előfordulásának megkülönböztetéséhez az *akció* fogalmát használjuk. Ez egy $(i, j)$ pár, amit $i^j$ alakban írunk, ahol $i$ a $j$-edik lépésben végrehajtott utasítás sorszáma. A példánkban a $12^{15}$ akció tartozik a kiírató utasításhoz az előző input esetén.

A *dinamikus szeletek* első precíz definícióját Korel és Laski adta meg ([13]). Cikkükben a dinamikus szeletet olyan futtatható programként definiálják, ami megkapható az eredeti programból utasítások törlésével. Szükséges, hogy egy adott inputra a dinamikus szelet és az eredeti program ugyanazt az értéket számolja ki egy $v$ változóra valamely kiválasztott végrehajtási lépésben.

A *dinamikus szeletelési feltételt* definiálhatjuk egy $(\mathbf{x}, i^j, V)$ hármasként, ahol $\mathbf{x}$ az input, $i^j$ egy akció az $EH$-ban, $V$ pedig változók egy halmaza. Egy szeletelési feltételre a dinamikus szelet definiálható azon utasítások halmazaként, melyek befolyásolhatják a $V$ halmazbeli változók értékét. Ez az eredetitől kissé eltérő definíció gyakran eredményez olyan utasításhalmazt, ami része ugyan az eredeti programnak, de önmagában nem futtatható.

Agrawal és Horgan a dinamikus szeleteket a következő módon definiálta ([3]): Adott egy $P$ program $t$ tesztesetéhez tartozó $H$ végrehajtási útja és egy $v$ változó. A $P$ program $H$-hoz és $v$-hez tartozó dinamikus szelete azon $H$-beli utasítások halmaza, melyeknek hatása van a $v$ változónak a végrehajtás végére kiszámított értékére.

Amellett, hogy ez a definíció egyetlen $v$ változóra korlátozódik, a fő különbség az, hogy a változó értékét a végrehajtás végén vizsgálják. Tehát nem érdekel, hogy mi a $v$ változó értéke a végrehajtás során, kivéve a legutolsó alkalmat. A hibakereséshez ez a definíció megfelelőbb a többinél.

Agrawal és Horgan kidolgoztak egy új módszert, ami dinamikus függőségi gráfot (DDG – Dynamic Dependence Graph) használ azért, hogy figyelembe tudja venni, egy adott utasítás különböző előfordulásaira más–más utasítások lehetnek hatással a változók újradefiniálása miatt. A DDG-ben külön csúcs van egy utasítás minden $EH$-beli előfordulásához. Ennek a gráfnak a használatával pontos dinamikus szelet állítható elő. A DDG legnagyobb hátránya a mérete. A DDG csúcsainak száma megegyezik a végrehajtott utasítások számával, amire nincs felső korlát. Az algoritmus memóriaigényének csökkentésére Agrawal és Horgan javasoltak egy eljárást, ami a DDG csúcsainak a számát csökkenti. Az eljárás lényege, hogy csak akkor szúrnak be új csúcsot a gráfba, ha ennek hatására új dinamikus szelet keletkezhet. Így a redukált gráf méretére egy felső korlát a különböző dinamikus szeletek száma, ami a legrosszabb esetben $O(2^n)$ ([19]), ahol $n$ az utasítások száma.

Ha kiszámoljuk a $(\langle \texttt{a=0}, \texttt{b=0}\rangle, 12^{15}, \texttt{s})$ szeletelési feltételhez tartozó szeletet az előző módszer alapján, a 2. ábrán látható eredményt kapjuk.

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

2. ábra. A keretezett utasítások alkotják a dinamikus szeletet

# 3. DINAMIKUS SZELETEK „FORWARD” SZÁMÍTÁSA

Az egyszerűség kedvéért először bemutatjuk a Gyimóthy Tibor és társai által egyszerű programnyelvre kidolgozott algoritmust, majd a 4. fejezetben ennek a C nyelvre történő kibővítését írjuk le.

A Gyimóthy Tibor és munkatársai által kidolgozott algoritmus előrehaladó („forward”), ami azt jelenti, hogy a szükséges információkat (azaz egy adott utasításhoz tartozó dinamikus szeletet) az utasítás lefutásával egyidőben számolja ki. Ennek következtében az eljárás globális, azaz miután az utolsó utasítás is végrehajtódott, az összes végrehajtott utasításhoz tartozó szeletet megkapjuk. A globális szeletelés nem szükséges a hibakereséshez, de hasznos lehet a teszteléshez.

Az algoritmushoz nem szükséges a DDG. Ehelyett azon utasítások halmazát számolják ki és tárolják el, amelyek hatással vannak az éppen végrehajtott utasításra. Ezen a módon elkerülik a felesleges információk kiszámítását.

Az algoritmus leírása előtt ismertetünk néhány - általuk is használt - alapfogalmat és jelölést. Alapvetően a [13] cikkre támaszkodtak, de néhány helyen elvégezték a szükséges módosításokat.

A dinamikus szeletelés fogalmait az 1. ábrán látható példaprogramra alkal-

mazva mutatjuk be.

Gyimóthy Tibor és munkatársai egy olyan programábrázolást alkalmazott, ami csak a változók definícióját (definition, d) és használatát (use, U), illetve a közvetlen vezérlési függőségeket tudja tárolni. Erre az ábrázolásra mint *D/U programábrázás* fogunk hivatkozni. Az eredeti program egy utasításához a következő D/U kifejezés tartozik:

$$i.\ d : U,$$

ahol $i$ az utasítás sorszáma, $d$ pedig az a változó, amelyik új értéket kap, ha az utasítás értékadó utasítás. Kiírató utasítás vagy predikátum esetén $d$ egy újonnan létrahozott „kimeneti változó"-nevet vagy „predikátum-változó"-nevet jelöl (lásd az alábbi példát). Az $U$ halmaz változók egy halmaza úgy, hogy $U = \{u_1, u_2, ..., u_n\}$ esetén minden $u_k \in U$ egy, az $i$ utasításban használt változó vagy egy predikátum-változó, amitől az $i$ utasítás (közvetlenül) kontrol-függ. (Ha az *entry* kezdőutasítást definiáljuk, minden $U$ halmazban pontosan egy predikátum-változó lesz.)

A példánk D/U ábrázolása a következő:

| $i.$ | $d:$ | $U$ |
|---|---|---|
| 1. | $n:$ | $\emptyset$ |
| 2. | $a:$ | $\emptyset$ |
| 3. | $i:$ | $\emptyset$ |
| 4. | $s:$ | $\emptyset$ |
| 5. | $p5:$ | $\{a\}$ |
| 6. | $s:$ | $\{p5\}$ |
| 7. | $p7:$ | $\{i,n\}$ |
| 8. | $p8:$ | $\{p7,a\}$ |
| 9. | $s:$ | $\{s,p8\}$ |
| 10. | $s:$ | $\{s,p8\}$ |
| 11. | $i:$ | $\{i,p7\}$ |
| 12. | $o12:$ | $\{s\}$ |

3. ábra. A példaprogram D/U ábrázolása

A $p5$, $p7$ és $p8$ predikátum-változókat jelölnek, az $o12$ pedig kimeneti-változót, aminek az értéke a kiírató utasításban használt változó(k)tól függ.

Ezek után a program egy adott bemenethez tartozó dinamikus szeletét a bemenethez tartozó végrehajtási út és a program D/U ábrázolása alapján a következőképpen számítják ki. A végrehajtási út minden utasítását az elsőtől kezdve sorban feldolgozzák. Egy $i.\ d : U$ utasítás feldolgozása közben kiszámítják a $DynSlice(d)$ halmazt, mely az összes olyan utasítást tartalmazza, ami hatással van a $d$ változó $i$ utasításbeli értékére. A D/U programábrázolás használatával az adat- és kontrolfüggőségek azonos módon kezelhetők. Miután egy utasítás végrehajtódott és a megfelelő $DynSlice$ halmazt kiszámolták, meghatározzák az $LS(d)$-t, ami a $d$ változó utolsó definíciójának a helye. Nyilvánvaló, hogy a

$j$-edik lépésben végrehajtott $i.\ d : U$ utasítás után az $LS(d)$ értéke mindaddig $i$ marad, amíg egy következő utasításban a $d$-t újra nem definiáljuk. Ha $p$ predikátum, $LS(p)$ a predikátum legutolsó kiértékelésének a helyét jelenti. Ha például $EH(10) = 7$ (azaz az aktuális akció $7^{10}$), akkor $LS(d) = 7$.

A dinamikus szeleteket a következő módon számíthatjuk ki. Tegyük fel, hogy a $t$ bemeneten futtatunk egy programot. Miután az $i.\ d : U$ utasítás a $p$ lépésben végrehajtódott, a $DynSlice(d)$ halmaz pontosan azokat az utasításokat fogja tartalmazni, amelyek benne vannak a $C = (t, i^p, U)$ szeletelési feltételhez tartozó szeletben. A $DynSlice$ halmazok a következő egyenlőség alapján számíthatók:

$$DynSlice(d) = \bigcup_{u_k \in U} \Big( DynSlice(u_k) \cup \{LS(u_k)\} \Big)$$

Miután a $DynSlice(d)$ halmazt kiszámoltuk, meghatározzuk $LS(d)$ értékét az értékadó utasításokra és predikátumokra:

$$LS(d) = i$$

Megjegyezzük, hogy ez a kiszámítási sorrend kötött, mert a $DynSlice(d)$ számításánál egy előző végrehajtási lépésben kiszámolt $LS(d)$ értékre van szükség, és nem az újonnan kiszámoltra (ez jól látszik például a ciklusbeli `x = x + y` utasítás esetén).

Látható, hogy a dinamikus szelet számítása közben nem használnak dinamikus függőségi gráfot, csak a D/U programábrázolást, ami kevesebb helyet foglal, mint az eredeti forráskód, és a fenti módszer ugyanazt a szeletet számolja, mint az [4] cikkben leírt DDG-t használó eljárás.

Az algoritmus formalizálva a 4. ábrán látható.

**program** DynamicSlice
**begin**
  $LS$ és $DynSlice$ halmazok inicializálása
  D/U felépítése
  $EH$ kiszámítása
  **for** $j = 1$ **to** $EH$ *elemszáma*
    a D/U aktuális eleme $i^j.\ d :\ U$
    $DynSlice(d) = \bigcup_{u_k \in U} \big( DynSlice(u_k) \cup \{LS(u_k)\} \big)$
    $LS(d) = i$
  **endfor**
  Output: $LS$ és $DynSlice$ halmazok az összes változó utolsó definíciójához
**end**

4. ábra. Dinamikus szeletelés algoritmusa

Az $EH$ az eredeti kód instrumentálásával, majd az instrumentált program futtatásával kapható meg. A instrumentálásról a 4. fejezetben lesz szó.

A fenti módszert alkalmazva az 1. ábrán látható példára, az $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 \rangle$ végrehajtási út esetén a következő értékeket kapjuk:

| Akció | $d$ | $U$ | $DynSlice(d)$ | $LS(d)$ |
|---|---|---|---|---|
| $1^1$ | $n$ | $\emptyset$ | $\emptyset$ | 1 |
| $2^2$ | $a$ | $\emptyset$ | $\emptyset$ | 2 |
| $3^3$ | $i$ | $\emptyset$ | $\emptyset$ | 3 |
| $4^4$ | $s$ | $\emptyset$ | $\emptyset$ | 4 |
| $5^5$ | $p5$ | $\{a\}$ | {2} | 5 |
| $7^6$ | $p7$ | $\{i, n\}$ | {1,3} | 7 |
| $8^7$ | $p8$ | $\{p7, a\}$ | {1,2,3,7} | 8 |
| $10^8$ | $s$ | $\{s, p8\}$ | {1,2,3,4,7,8} | 10 |
| $11^9$ | $i$ | $\{i, p7\}$ | {1,3,7} | 11 |
| $7^{10}$ | $p7$ | $\{i, n\}$ | {1,3,7,11} | 7 |
| $8^{11}$ | $p8$ | $\{p7, a\}$ | {1,2,3,7,11} | 8 |
| $10^{12}$ | $s$ | $\{s, p8\}$ | {1,2,3,4,7,8,10,11} | 10 |
| $11^{13}$ | $i$ | $\{i, p7\}$ | {1,3,7,11} | 11 |
| $7^{14}$ | $p7$ | $\{i, n\}$ | {1,3,7,11} | 7 |
| $12^{15}$ | $o12$ | $\{s\}$ | {1,2,3,4,7,8,10,11} | 12 |

A végső szelet a $DynSlice(o12)$ és az $\{LS(o12)\}$ uniója.

# 4. VALÓDI C PROGRAMOK DINAMIKUS SZELETELÉSE

Az előző fejezetben bemutattuk a témavezetőnk és társai által kidolgozott szeletelési módszert. Az eljárást egy egyszerű programnyelvre dolgozták ki, így C nyelvű programokra csak jelentős megszorításokkal alkalmazható (nincs függvényhívás, csak skalár változók használhatók és az értékadó operátort utasításként kell kezelni). Ebben a fejezetben ezt az algoritmust bővítjük ki úgy, hogy alkalmas legyen valódi C programok szeletelésére is. Ez azt jelenti, hogy meg kell oldanunk többek közt a *mutatók*, a *függvényhívások* és az *ugró utasítások* problémáját.

A mutatók kezelésének szükségessége miatt egy kicsit változtatnunk kellett a szeletelési feltétel jelentésén. Ez ezt jelenti, hogy ha például egy `*p` pointer dereferencia dinamikus szeletére vagyunk kíváncsiak, akkor egy *memóriacím* dinamikus függőségeit keressük (és nem csupán egy változóét, mint az eredeti definícióban). (Mint azt látni fogjuk, a `*p` dereferencia szelete magába foglalja a `p` pointer és az általa mutatott memóriacím függőségeit is.)

A tömbök és a struktúrák mezőinek kezelése visszavezethető memóriacímek szeletelésére (hasonlóan a mutatókhoz).

A C programok szeletelésére alkalmas eljárásunk a következő fő lépésekből áll:

- A program statikus függőségei alapján elkészítjük annak D/U ábrázolását, és instrumentáljuk a programot, hogy a szükséges futás közbeni információkat kinyerhessük belőle.

- Az instrumentált programot lefordítjuk és futtatjuk, így kapunk egy olyan fájlt, ami a szeletelő algoritmushoz szükséges dinamikus információkat tartalmazza. Ennek neve: $TRACE$.
- Az algoritmust végrehajtjuk az előzőleg megkapott D/U és $TRACE$ segítségével.

Az előző fejezetben az $i$ utasítás D/U ábrázolását $i.\ d : U$-ként definiáltuk. C programokra a D/U ábrázolás $d :$ *$U$ elemek* egy *sorozatát* fogja tartalmazni, azaz:

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

Ez azért szükséges, mert egy C utasításban (azaz kifejezésben) több memóriahely is új értéket kaphat. Megjegyezzük, hogy a sorozat tagjainak a sorrendje fontos, hiszen előző D/U elemek $d$ értékei használhatók következő D/U elemek $U$ halmazaiban. Az elemek sorrendjét a kifejezések ,,végrehajtási" (kiértékelési) sorrendje[1] határozza meg.

A definiált $d$ és a használt $u_k \in U$változóknak többféle jelentése lehet. Ezek a következők:

- Skalár változók. Ezek az ,,általános" globális vagy lokális változók (a címük egy scope-on belül nem változik).
- Predikátum változók. Jelölésük $p_n$, ahol $n$ annak az utasításnak a sorszáma, ahol $p$-t *definiáljuk* (lásd az előző fejezetet).
- Címke változók. Jelölésük $l_n$, ahol $n$ a címkézett utasítás sorszáma. Ezeket a változókat az ugró utasítások (`goto`, `case`, `break`, `continue`) kezelésére használjuk. A címke változót a címkézett utasítás függvényén belül a címkét követő összes utasításban *használjuk*, és az erre a címkére ugró utasításokban *definiáljuk*. Egy $U$ halmazban egyszerre több címke változó is szerepelhet (ellentétben a predikátum változókkal, melyekből fegfeljebb egy lehetséges).
- Kimeneti változók. Jelölésük $o_n$, ahol $n$ az utasítás sorszáma. Definíció szerint ezek a változók egyfajta ,,nem létező" változók, amik azokon a helyeken generálódnak, ahol az $U$ halmaz nem üres, de semmilyen változó nem kap értéket. Ilyen például az, ha egy függvényt eljárásként hívunk (nem vesszük el a visszatérési értékét), vagy az egyszerű, mellékhatás nélküli kifejezés-utasítás, illetve az egyszerűség kedvéért néhány kiírató utasítás (mint a `printf`).
- Dereferencia változók. Jelölésük $d_n$, ahol $n$ egy egyedi számláló minden dereferencia előfordulásra. Ezt a változót akkor használjuk, amikor egy memóriahely értékét *definiáljuk* vagy *használjuk* egy mutatón (tömbön vagy egy struktúra mezőjén) keresztül.

[1]a C nyelvben ez a sorrend nem mindig rögzített, ilyen esetekben a használt elemző által megadott sorrendet vesszük figyelembe

- Argumentum változók. Jelölésük $arg(f, n)$, ahol $f$ a függvény neve, $n$ pedig a függvény argumentumának (paraméterének) a sorszáma. Az argumentum változót a függvényhívás helyén *definiáljuk*, és a hívott függvény belépési pontján *használjuk*.
- Return változók. Jelölésük $ret(f)$, ahol $f$ a függvény neve. A return változót a függvény (összes) kilépési pontján *definiáljuk*, és a függvényhívás helyén, mint a függvény értékét *használjuk*.

Az ugró utasítások kezelésére a címke változókat használjuk. Először a `goto` utasítás kezelését mutatjuk be, majd a `case`, `break` és `continue` utasításokat visszavezetjük erre. Az $n$ sorszámú utasításra ugró `goto` utasítások mindegyike *definiálja* az $l_n$ címke változót. Az $l_n$ változót az $n$ sorszámú és az azt követő, de vele egy függvényben levő összes utasítás *használja*. Ez ezért szükséges, mert a címkét követő összes utasítás (nem feltétlenül közvetlenül) kontrol–függ az ugró utasításban *használt* predikátum változótól. A címke változók ezeket a függőségeket mutatják.

A `break` és `continue` utasítások ciklikus vezérlési szerkezetek (`for`, `while` és `do-while`) belsejében visszavezethetők a `goto` utasításra. A `break` ekvivalens egy olyan `goto` utasítással, ami a ciklus blokkját követő első utasításra ugrik. A `continue` utasítás pedig egy olyan `goto`, ami a ciklus feltételének kiértékelésére (`while` és `do-while`) vagy a növelő kifejezésre (`for`) ugrik. Ezt úgy kezeljük, mintha az előtt az utasítás előtt, ahová ugrunk, egy címke lenne, az ugrás pedig egy `goto` hatására történne. Így a `break` vagy `continue` utasítás egy címke változót *definiál*, amit a függvényben a ciklus blokkjának végétől illetve a növelő utasítástól kezdve minden utasításban használunk.

A `switch` szerkezet blokkjában a `case` címkeként viselkedik, de egyszerűbben is meg lehet oldani a problémát. A `switch` utasításban *definiálunk* egy predikátum változót, amit az utasítás blokkján belül *használunk* (ahogy például egy `if` vagy `while` blokkjában tennénk). A `break` itt is visszavezethető egy olyan `goto` utasításra, ami kiugrik a `switch` blokkjából a blokkot követő első utasításra.

Az argumentum és return változók a függvényhívások kezeléséhez szükségesek. A függvényhívás helyén az $f$ függvény minden paraméterére *definiálunk* egy $arg(f, n)$ változót, ahol $n$ a paraméter sorszáma. Ezek a változók a függvény nekik megfelelő aktuális paraméterét fogják *használni*. Az $f$ függvény definíciójánál ezeket az argumentum változókat fogják használni a függvény „skalár" argumentumai (ez lesz a függvény első „utasítása"). Így amikor a szeleteket számoljuk, nem kell az argumentumok $U$ halmazait cserélgetni, hiszen az argumentum változókon keresztül mindig az aktuális függéseket kapjuk meg. Hasonló okok miatt vezetjük be a return változókat. A $ret(f)$ változót az $f$ függvény `return` utasításaiban *definiáljuk*, és a függvényhívás helyén a függvény értékeként *használjuk*.

A 5. ábrán egy C programot, és a hozzá tartozó statikusan kiszámolt D/U ábrázolását láthatjuk a fentebb említett jelölésekkel.

A dinamikus szelet kiszámításához a program statikus D/U ábrázolása mellett szükségünk van dinamikus információkra is. Ezeket az információkat az

| $i.$ | | $\langle d:\ U\rangle$ |
|---|---|---|
| | `#include <stdio.h>` | |
| | `int a, b;` | |
| | | |
| 1. | `int f(int x,int y) {` | $x:\{arg(f,1)\}$, $y:\{arg(f,2)\}$ |
| 2. | `  a += x;` | $a:\{a,x\}$ |
| 3. | `  b += y;` | $b:\{b,y\}$ |
| 4. | `  return x+2;` | $ret(f):\{x\}$ |
| | `}` | |
| | | |
| 5. | `int g(int y) {` | $y:\{arg(g,1)\}$ |
| 6. | `  a += y;` | $a:\{a,y\}$ |
| 7. | `  return y+1;` | $ret(g):\{y\}$ |
| | `}` | |
| | | |
| | `void main() {` | |
| | `  int s, *p;` | |
| 8. | `  s = 0;` | $s:\emptyset$ |
| 9. | `  scanf("\%d", &a);` | $a:\emptyset$ |
| 10. | `  scanf("\%d", &b);` | $b:\emptyset$ |
| 11. | `  p = &b;` | $p:\emptyset$ |
| 12. | `  while (*p < 10) {` | $p12:\{p,d1\}$ |
| 13. | `    s += f(3,4);` | $arg(f,1):\{p12\}$, $arg(f,2):\{p12\}$, $s:\{s,ret(f),p12\}$ |
| 14. | `    s += g(3);` | $arg(g,1):\{p12\}$, $s:\{s,ret(g),p12\}$ |
| | `  }` | |
| 15. | `  printf("\%d", *p);` | $o15:\{d2\}$ |
| 16. | `  printf("\%d", s);` | $o16:\{s\}$ |
| | `}` | |

5. ábra. Egy példa C program és a (statikus) D/U ábrázolása

eredeti program *instrumentálásával* kaphatjuk meg[2]. Az instrumentálást úgy végezzük, hogy fordítás után az instrumentált program futása csak annyiban térjen el az eredetitől, hogy a szükséges dinamikus információkat kiírja a $TRACE$ állományba. A $TRACE$ tartalmazza magát a végrehajtási utat és különféle ,,adminisztratív" információkat, mint például a skalár változók címei, függvény és blokk kezdet/vég, stb.

A $TRACE$ a következő sorokból áll össze:

[2] Alapvetően kétféle lehetőség van egy program instrumentálására: forrás-szintű és kód-szintű. Mi a forrás-szintű instrumentálást választottuk, többek közt a platformfüggetlenség biztosítása céljából és mert a szeletelési egységeink a forrás sorai. A kód-szintű instrumentálás az instrumentált kód gyorsabb futását és a könyvtári és rendszerfüggvények pontosabb kezelését eredményezheti [20].

- Végrehajtás út sorok ($EH$ sorok). Ezen sorok mindegyike egy-egy akciót tartalmaz. Ha a $TRACE$ többi sorát töröljük, akkor megkapjuk a program végrehajtási útját.
- Deklarációs sorok. Ezen sorok mindegyike egy-egy skalár változó nevét és címét tartalmazza. Két típusa van: globális és lokális, hiszen ezek kezelése némileg eltér egymástól. Ezek a sorok segítenek memóriacímekké alakítani a skalár változókat.
- Pointer sorok. Ezen sorok mindegyike egy-egy dereferencia változó nevét, értékét tartalmazza és azt, hogy éppen *definiáljuk* vagy *használjuk* a változót. Ezen sorok segítségével tudjuk a dereferencia változókat memóriacímekké alakítani.
- Függvény és blokk sorok. Ezek a sorok jelzik egy-egy függvény vagy blokk kezdetét/végét. Mivel különböző blokkokban azonos néven több változó is lehet, a változók címeit egy többszörös veremben tároljuk. Ennek a veremnek a pontos kezeléséhez kellenek ezek a sorok.

Miután a program statikus D/U abrázolását és a $TRACE$ fájlt is előállítottuk, ki tudjuk számítani a szeletelési feltételnek megfelelő programszeletet. A számolás a $TRACE$ alapján megy, ennek a sorait dolgozzuk fel különböző módon az elejétől kezdve. Ha a $TRACE$ aktuális eleme egy „adminisztratív" elem, akkor végrehajtjuk a szükséges módosításokat a tárolt adatokon (ilyen például a skalár változók címeit tartalmazó kettős verem karbantartása). Ha az elem egy $EH$ elem, akkor a hozzá tartozó $i$. $\langle d : U \rangle$ sorozatot dolgozzuk fel a következő módon. Minden a sorozatban szereplő D/U elemre kiszámoljuk a neki megfelelő $d'$ és $u'_k \in U'$ „dinamikus függőségeket", majd meghatározzuk a $DynSlice(d')$ halmazt, ami tartalmazza az összes utasítást, ami az $i$ utasítás végrehajtásakor hatással van $d'$ értékére, és az $LD(d')$-t a következőképpen:

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

$$LS(d') = i$$

Mint az előző fejezetben, a kiszámítási sorrend itt is kötött.

Függvényhívás esetén az aktuális D/U sorozat nem dolgozható fel egy egyszerű ciklusban, ami sorra veszi a sorozat elemeit. Ilyen esetekben el kell tárolni egy veremben, hogy hol tartunk az aktuális sorozat feldolgozásában, és miután a függvény visszatért, a feldolgozást innen kell folytatni.

A $d'$ és $u'_k$ értékeit a $d$ és $u_k$ statikusan kiszámolt értékeiből kapjuk a következő módon ($u'_k$ számítása $u_k$-ból ugyanígy történik):

- ha $d$ egy *skalár változó*, akkor $d'$ a memóriacíme lesz (ami a $TRACE$ alapján a kettős veremből meghatározható),
- ha $d$ egy *dereferencia változó*, akkor $d'$ a neki megfelelő pointer értéke (ez is memóriacím) lesz (ismét a $TRACE$ alapján).

- ha $d$ *predikátum változó*, akkor $d'$-t a megfelelő $p_n$ predikátum változóból kapjuk úgy, hogy kiegészítjük egy „mélységi" információval, ami a *függvényhívási verem* mélysége, és $pn(k)$-val jelöljük (erre a rekurzív hívások miatt van szükség, ahol ugyanaz a predikátum a függvény két különböző hívásához tartozhat),
- minden más esetben $d$ és $d'$ megegyezik.

A 6. ábrán formalizálva megadjuk a C programok szeletelésére alkalmas algoritmust.

Most a fenti eljárást alkalmazzuk a 5. ábrán látható példaprogramunkra és az $(\langle$`a=2`, `b=6`$\rangle, 15^{14},$ `*p`$)$ szeletelési feltételre. Erre az inputra a következő végrehajtási utat kapjuk: $\langle 8, 9, 10, 11, 12, 13/1, 2, 3, 4/13, 14/5, 6, 7/14, 12, 15, 16\rangle$. (A „kettős" $EH$ elemek, azaz a 13/1, 4/13, 14/5 és 7/14 a függvényhívás/visszatérés és paraméterátadás „virtuális utasításai".) A végrehajtás alatt a következő értékeket kapjuk:

| Action | $d'$ | $U'$ | $DynSlice(d')$ | $LS(d')$ |
|---|---|---|---|---|
| $8^1$ | 6684144 | $\emptyset$ | $\emptyset$ | 8 |
| $9^2$ | 4347824 | $\emptyset$ | $\emptyset$ | 9 |
| $10^3$ | 4347828 | $\emptyset$ | $\emptyset$ | 10 |
| $11^4$ | 6684148 | $\emptyset$ | $\emptyset$ | 11 |
| $12^5$ | $p12(1)$ | {6684148, 4347828} | {10,11} | 12 |
| $13^6$ | $arg(f,1)$ | $\{p12(1)\}$ | {10,11,12} | 13 |
| $13^6$ | $arg(f,2)$ | $\{p12(1)\}$ | {10,11,12} | 13 |
| $1^6$ | 6684060 | $\{arg(f,1)\}$ | {10,11,12,13} | 1 |
| $1^6$ | 6684064 | $\{arg(f,2)\}$ | {10,11,12,13} | 1 |
| $2^7$ | 4347824 | {4347824, 6684060} | {1,9,10,11,12,13} | 2 |
| $3^8$ | 4347828 | {4347828, 6684064} | {1,10,11,12,13} | 3 |
| $4^9$ | $ret(f)$ | {6684060} | {1,10,11,12,13} | 4 |
| $13^9$ | 6684144 | $\{6684144, ret(f), p12(1)\}$ | {1,4,8,10,11,12,13} | 13 |
| $14^{10}$ | $arg(g,1)$ | $\{p12(1)\}$ | {10,11,12} | 14 |
| $5^{10}$ | 6684064 | $\{arg(g,1)\}$ | {10,11,12,14} | 5 |
| $6^{11}$ | 4347824 | {4347824, 6684064} | {1,2,5,9,10,11,12,13,14} | 6 |
| $7^{12}$ | $ret(g)$ | {6684064} | {5,10,11,12,14} | 7 |
| $14^{12}$ | 6684144 | $\{6684144, ret(g), p12(1)\}$ | {1,4,5,7,8,10,11,12,13,14} | 14 |
| $12^{13}$ | $p12(1)$ | {6684148, 4347828} | {1,3,10,11,12,13} | 12 |
| $15^{14}$ | $o15$ | {4347828} | {1,3,10,11,12,13} | 15 |
| $16^{15}$ | $o16$ | {6684144} | {1,4,5,7,8,10,11,12,13,14} | 16 |

**program** DynamicSliceForC
**begin**
  $LS$ és $DynSlice$ halmazok inicializálása
  D/U felépítése
  $TRACE$ kiszámítása
  aktuális D/U elem = nil
  **for** *a* $TRACE$ *minden sorára*
    **case** *a* $TRACE$ *aktuális sora* **of**
      *függvénykezdet-jel*:
        push(aktuális D/U elem)
        aktuális D/U elem = nil
      *függvényvég-jel*:
        pop(aktuális D/U elem)
      *EH elem*:
        a jelenlegi akció az $EH$-ban $i^j$
        aktuális D/U elem = az $i$. $\langle d : U\rangle$ első eleme
      *más:*
        az aktuális D/U elemben előforduló feloldatlan
        memóriacím-referenciák feloldása
    **endcase**
    **while** az aktuális D/U elem feldolgozható*
      $d'$ és $U'$ kiszámítása az aktuális D/U elem alapján
      $DynSlice(d') = \bigcup_{u'_k \in U'} \big(DynSlice(u'_k) \cup \{LS(u'_k)\}\big)$
      $LS(d') = i$
      aktuális D/U elem = az $i$. $\langle d : U\rangle$ következő eleme
    **endwhile**
  **endfor**
  Output: $LS$ és $DynSlice$ halmazok az összes használt
  memóriacím utolsó definíciójához
**end**

*Ez akkor **igaz** ha: az (aktuális D/U) *elem* pozíciójában a statikus D/U alapján nincs függvényhívás **és** az *elem* $\neq$ nil **és** az *elem* nem tartalmaz feloldatlan memóriacím-referenciát.

6. ábra. Dinamikus szeletelő algoritmus C programokra

A végső szelet megkapható a $DynSlice(o15)$ és az $\{LS(o15)\}$ uniójaként. A végeredmény a 7. ábrán látható. A szelet az első oszlopban ponttal megjelölt sorokat tartalmazza. észrevehető, hogy a dinamikus szelet azokat az utasításokat tartalmazza, amik befolyásolták a `4347828`-as, a `p` által mutatott memóriahely értékét (ami valójában a `b` skalár változó értéke).

Megfigyelhető az is, hogy az $(\langle$`a=2`$,$`b=6`$\rangle, 16^{15},$ `s`$)$ szeletelési feltételhez tartozó dinamikus szelet (második oszlop) csak azokat az utasításokat tartalmazza, amik az `s` skalár változó értékét befolyásolják, azaz amik a két globális értékére

vannak hatással, nincsenek benne a szeletben, hiszen az `s` csak a két függvény visszatérési értékét használja (amik pedig konstans értékektől függenek). Megjegyezzük, hogy ennek a második szeletnek a kiszámításához nem volt szükség az algoritmus újbóli lefuttatására, hiszen az eljárás ,,globális" egy adott inputra, és az összes változó szeletét kiszámolja egyszeri lefutás során.

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

7. ábra. Az `a=2` és `b=6` inputra kiszámolt dinamikus szeletek

# 5. ÖSSZEFOGLALÁS

Különféle szeletelési módszereket alkalmaznak hibakeresésre, tesztelésre és karbantartásra. A szeletelő algoritmusok lehetnek statikus vagy dinamikus szeletelő eljárások. Bizonyos alkalmazásokban, mint például a hibakeresés, a dinamikus szeletelés sokkal eredményesebb mint a statikus. A [20] cikkben leírt tapasztalatok azt mutatják, hogy a dinamikus szelet a végrehajtott utasítások

kevesebb, mint a felét tartalmazza, illetve nagy valószínűséggel az egész program méretének a 20%-a alatt marad.

Sokféle dinamikus szeletelési módszert publikáltak már, de a legtöbbjükben a dinamikus függőségi gráfot (DDG) használták a program futásának belső ábrázolására. Ennek a legnagyobb hátránya az, hogy a DDG méretére nincs korlát, hiszen minden végrehajtott utasítás külön csomópontba kerül.

Gyimóthy Tibor és munkatársai kidolgoztak egy algoritmust, ami a program D/U ábrázolását használja a DDG helyett, így alkalmas nagyobb méretű programok szeletelésére is. Az eljárás a program futásával párhuzamosan számolja ki a hozzá tartozó szeletet. Az algoritmus azonban „csak" egy egyszerű programnyelvre lett kidolgozva.

Ebben a dolgozatban ezt az algoritmust bővítettük ki úgy, hogy alkalmas legyen valódi C programok dinamikus szeletelésére. Megoldottunk néhány, a C nyelvben előforduló problémát (mint például a pointerek kezelése, függvényhívások, ugró utasítások). Az algoritmus fő előnye, hogy valódi méretű C programokra is alkalmazható, hiszen a memóriaigénye a program által használt különböző memóriahelyek (és nem a végrehajtott utasítások) számával arányos.

A rendszerünket teszteltük néhány egyszerű programon, aminek az eredményét az alábbi táblázat mutatja:

| | Sorok száma[1] | EH mérete | Memóriahelyek[2] |
|---|---|---|---|
| Teszt #1 | 68 | 13524 | 175 |
| Teszt #2 | 133 | 3673 | 232 |
| Teszt #3 | 246 | 9322 | 433 |

# Hivatkozások


[1] Agrawal, H., DeMillo, R.A., and Spafford, E.H. : Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)* (1991), pp. 60-73.

[2] Agrawal, H., DeMillo, R. A., and Spafford, E.H. : Debugging with dynamic slicing and backtracking. *Software—Practice And Experience*, 23(6):589-616, June 1993.

[3] Agrawal, H., and Horgan, J. : Dynamic program slicing. *SIGPLAN Notices*, No. 6, 1990, pp. 246-256.

[4] Agrawal, H., Horgan, J.R., Krauser, E.W., and London, S.A. : Incremental Regression Testing. *Proceedings of the IEEE Conference on Software Maintenance*, Montreal, Canada,1993.


[1] A végrehajtható utasítások száma a programban

[2] A program által használt különböző memóriahelyek száma

[5] Beck, J., and Eichmann, D. : Program and Interface Slicing for Reverse Engineering. In *Proc. 15th Int. Conference on Software Engineering*, Baltimore, Maryland, 1993. IEEE Computer Society Press, 1993, 509-518.

[6] Forgács, I. : An exact array reference analysis for data flow testing. In *Proc. of the 18th Int. Conference on Software Engineering*, Berlin 1996, 565-574.

[7] Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimóthy, T. : Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems 1*, 4 (1992), 303-322.

[8] Gallagher, K.B., and Lyle, J.R. : Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering* 17, 8, 1991, 751-761.

[9] Gyimóthy, T., Beszédes, Á, and Forgács, I. : An Efficient Relevant Slicing Method for Debugging. In *Proc. 7th European Software Engineering Conference (ESEC)*, Toulouse, France, Sept. 1999. LNCS 1687, pages 303-321.

[10] Horwitz, S., Reps, T., and Binkley, D. : Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26-61.

[11] Kamkar, M. : An Overview and Comparative Classification of Program Slicing Techniques. *J.Systems Software* 31:197-214,1995.

[12] Kamkar, M. : Interprocedural Dynamic Slicing with Applications to Debugging and Testing, Ph.D. Thesis, Linkoping University, 1993.

[13] Korel, B., and Laski, J. : Dynamic slicing in computer programs. *The Journal of Systems and Software* vol. 13, No. 3, 1990, pp. 187-195.

[14] Korel, B., and Rilling, J. : Application of dynamic slicing in program debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97)*, Linkoping, Sweden, May 1997.

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

[16] Landi, W., and Ryder, B. : A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation* (San Francisco, 1992), pp. 235-248. *SIGPLAN Notices* 27(7).

[17] Ottenstein, K., and Ottenstein, L. : The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (1984), pp. 177-184. *SIGPLAN Notices* 19(5).

[18] Rothermer, G., and Harrold, M. J. : Selecting tests and identifying test coverage requirements for modified software. In *Proc. ISSTA'94* Seattle. 1994, 169-183

[19] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121-189, September 1995.

[20] Venkatesh, G. A. : Experimental Results from Dynamic Slicing of C Programs. In *ACM Transactions on Programming Languages and Systems*, Vol 17. No 2, March 1995, Pages 197-216.

[21] Weiser M. : Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4, 1984, 352-357.