Szegedi Tudományegyetem
Matematika- és Számítástudományok Doktori Iskola
Informatikai Doktori Program

Forráskód analízis és szeletelés a programmegértés támogatásához

PhD értekezés tézisei

Beszédes Árpád
SZTE Szoftverfejlesztés Tanszék

Témavezető:

Dr. Gyimóthy Tibor

Szeged
2004

„A számítógépek jók az utasításaink követésében, de rossz gondolatolvasók."
— Donald E. Knuth

Bevezetés

Ez az összefoglaló ismerteti a szerző tevékenységét és tudományos eredményeit a „Forráskód analízis és szeletelés a programmegértés támogatásához" c. PhD értekezésében. A szerző a szoftverfejlesztés tárgykörének meglévő rendszerek modellezését és analízisét célzó kutatási ágak területén végzett kutatásainak eredményeképpen módszereket, technológiákat és eszközöket dolgozott ki a programmegértés támogatására.

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

A kód értelmezésének igénye az életciklus szinte bármely pontján felmerülhet, kezdve a nyomkövetés utáni hibajavítással, amikor a hibás viselkedés okát próbáljuk felfedni. A hibajavítás a kód első megjelenésekor felmerül, de a verifikáció fázisában, a program tesztelésének eredményeképpen is. Továbbá, a fejlesztés után, a szoftver evolúciós fázisában a karbantartás alapvető tevékenység, amikor sok esetben a forráskód az egyetlen megbízható reprezentációja a rendszernek. Ez adódhat abból, hogy manapság a szoftverrendszerek rendkívül gyorsan változnak, újabb és újabb verziók megjelenését követeli meg a piac és a rohamosan fejlődő új technológiák. Az új verziók sokszor a meglévő rendszerekből jönnek létre evolúció által, ami a gyors fejlesztés miatt legtöbbször azt eredményezi, hogy a kódot kísérő magasabb szintű modellek hiányosak, nincsenek összhangban a kóddal, vagy teljes egészében hiányoznak. Ezen problémák eredményezték az *újratervezés* (reengineering) tudományág létrejöttét a szoftverfejlesztésen belül. Ezen tevékenység részeként első lépésként mindig szükség van a *meglévő rendszerek modellezésére* (reverse engineering), amikor is a rendelkezésre álló forráskódból megpróbáljuk meghatározni a rendszer komponenseit és az azok közötti összefüggéseket, továbbá előállítani egy rendszer magasabb absztrakciós szinten lévő, vagy más jellegű reprezentációját [4].

A fentiek miatt láthatjuk, hogy nagy igény van a forráskód megértését támogató eszközökre, hiszen a megértés folyamatának számos lépése mechanikus, melyet eszköz-támogatással egyszerűbbé, gyorsabbá tehetünk. Mint ahogy a fordítóprogram is képes előállítani különböző reprezentációkat a programról, melyekre szüksége van saját céljának eléréséhez, úgy automatikus, kódot *analizáló*

eszközök is kifejleszthetők, melyek egyéb reprezentációkat tudnak létrehozni. Az igazi haszon az, hogy a forráskód analízisével, különböző reprezentációk előállításával és egyéb kérdések megválaszolásával a forráskódról, ember által könnyebben felfogható formában tudjuk prezentálni a szoftverrendszert.

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

A fenti problémakört tűztük ki feladatunknak, amikor belevágtunk egy olyan technológia- és eszközrendszer kidolgozásába, amelynek néhány alapvető eleme máris működőképes, nemzetközi elismerésnek örvend és valós környezetben használatos. E hosszútávú fejlesztésünk jelenlegi eredménye egy általános analizáló keretrendszer, melynek neve Columbus [6, 7, 8, 9, 10], és a szükséges technológiák és eszközök. Jelenleg a C/C++ nyelveket céloztuk meg, hiszen ezen nyelvek a legelterjedtebbek között vannak és már elég régóta léteznek ahhoz, hogy egyre több, bennük íródott rendszernél legyen szükség újratervezésre, megértésre. Fejlesztésünk konkrét eredményei egy általános analizátor és séma C/C++ nyelvekre, valamit egyéb, forráskód analízisen alapuló eszközök, mint amilyen a dinamikus szeletelőnk [2, 3, 13]. Kísérleteink végzéséhez szükséges eszköztárunkban szerepel még néhány külső eszköz is, amelyek területeivel pillanatnyilag nem kívánunk foglalkozni. Ezek közé sorolhatók a különböző szoftver-vizualizáló eszközök, és egy statikus szeletelő is.

A szerző a programmegértést támogató környezetünk két fontos elemét dolgozta ki (ld. az 1. ábrát). Az egyik a minden analizálási feladat alapja, egy általános célú C/C++ forráskód analizátor, amely meghatározott sémának megfelelő modellt állít elő. Az analizátor a fordítóprogramok elemzőihez képest számos tekintetben speciális kellett hogy legyen. A másik eredmény a dinamikus programszeletelő, egy speciális kód-analizálási eszköz, amely sokrétű alkalmazási területekkel bír. Utóbbihoz két hatékony algoritmus és C nyelvre való megvalósítás tartozik, és ellentétben a korábbi módszerekkel, valódi alkalmazásokban is használható hatékonysága és részletekbe menő kidolgozottsága miatt.

Az eredmények négy tézisbe sorolhatók, amelyekből egy az első témához kapcsolódik és három a másodikhoz. Jelen összefoglaló is ennek megfelelően két részből áll, melyek végein megtalálhatók az eredmények tézisszerű összefoglalásai.

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

I.1. C/C++ forráskód analízis nehézségei

A C++ programozási nyelvet az egyik legkomplexebb nyelvként tartjuk nyilván, ami elsődlegesen abból adódik, hogy ötvözi az alacsony szintű programozás lehetőségeit (például közvetlen memória elérés és mutató aritmetika) a magas szintű paradigmákkal (objektumorientáltság, generikus programozás).[1] Ez természetesen előnyt jelent programozás-technológiai szempontból, ugyanakkor azt is eredményezi, hogy a C/C++ rendkívül nehezen analizálható. A nehézség adódik a bonyolult, kétértelműségekkel teli szintaxisból és a számos további szemantikai szabályból és egyéb előírásból, amelyek a nyelvet alkotják.

A forráskód feldolgozásának két alapvető célja van. Az egyik a kód lefordítása futtatható programmá a fordítóprogram segítségével, a másik pedig az általunk vizsgált kód-analízis. Utóbbit számos cél elérése érdekében végezzük a szoftverkarbantartás, a meglévő rendszerek utólagos modellezése (reverse engineering), dokumentálása, megértése és validálási alkalmazásokban. A két feldolgozás közös tulajdonsága, hogy szükség van egy komponensre, amely az adott nyelv szintaktikai és egyes szemantikai szabályainak megfelelően elemzi a forráskódot, és abból előállít egy magasabb szintű reprezentációt a további feldolgozás részére. E modul közismert elnevezése a *front end*, vagyis elülső rész. A fordítóprogramban használatos front end által előállított reprezentáció az absztrakt szintaxis fa és abból származtatott egyéb információk, amelyeket fel kell használni a tárgykód elkészítéséhez. A másik esetben is elő kell állítanunk valamilyen felsőbb szintű modellt, de ennek legtöbb esetben különbözőnek kell lennie a fordítóprogramban használatostól. Következésképpen, egy általános programmegértést és karbantartást támogató analizátorban speciális front end-et kell alkalmazni, amelynek sajátos tulajdonságokkal kell rendelkeznie. Az analizátorokban lévő front end-et szokták még *tény-feltárónak* (fact extractor) is nevezni.

Túl a nyelv helyes felismerésén, egy általános célú analizátor front end-jének számos olyan követelménynek kell eleget tennie, amelyek a fordítóprogramok esetében nem mindig játszanak szerepet. Ezért a fordítóprogramok által használt technológiák általában nem alkalmasak céljainkra. A [10] cikkben közöltünk egy követelménylistát, amelyek közül itt áttekintjük a legfontosabbakat:

- *Teljesség*. Az általánosság miatt a forráskód teljes analízisére van szükség, és nem elegendő az ún. *fuzzy* elemzési technika, amit számos tény-feltáró analizátor alkalmaz, hiszen ezek részleges reprezentációjára képesek csak a nyelvnek.

- *Részletesség*. A fordítóprogramokban alkalmazott elemzők és modellek tipikusan kihagyják az olyan információkat mint pl. vesszők, zárójelek, kommentek, redundáns deklarációk, stb. Néhány tény-feltáró alkalmazásban azonban az eredeti, forrás-hű információkra van szükség, ezért a front end gyakorlatilag a forráskódban található összes információt ki kell hogy gyűjtse.

- *Kapcsolódás*. Az elemzett forrásból előállított belső reprezentációban szereplő információ tetszőleges célra lesz felhasználva, ezért fel kell készülni az analizátor további részeivel való minél könnyebb kapcsolódásra.

- *Hibatűrés*. Egy általános célú front end-nek képesnek kell lennie a lehető legtöbb kinyerhető információt összegyűjteni akár nem teljes, esetleg hibás (nem fordítható) kódból is. Emiatt a fordítóprogramhoz képest jobb hibatűrési képességekkel kell rendelkeznie.

[1] A C++ a C nyelvet (annak procedurális elődjét) gyakorlatilag teljes egészében tartalmazza, így a továbbiakban a kettőt együtt tárgyaljuk.

- *Sebesség*. A sebesség kevésbé kritikus egy analizátornál, mint a fordítóprogram esetében, hiszen az utóbbit valószínűleg gyakrabban futtatjuk, főleg fejlesztési fázisban. Ugyanakkor, ne feledjük el, hogy egy karbantartási célt szolgáló analizátornak általában nagy mennyiségű forráskódot kell feldolgoznia egyhuzamban, tehát a sebesség kérdése nem teljesen elhanyagolható.

- *Előfeldolgozás*. C/C++ front end esetében az előfeldolgozást is nyilvánvalóan el kell végezni (például a makrók kifejtését vagy a fejlécfájlok feldolgozását). Ezen kívül az előfeldolgozóval kapcsolatos információkat is ki kell gyűjteni, hiszen azokra is szükség lehet (például *include*-hierarchia előállításához).

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

- *Felhasználói felület*. A legáltalánosabb használhatóságot az jelenti, ha az eszköz parancssori üzemmódban (is) tud működni, hiszen így tetszőleges eszközből behívható. Továbbá, a legtöbb analizálandó valós szoftverrendszert valószínűleg valamilyen automatikus fordítási környezet kísér, amelybe a legkönnyebb beépülést a parancssori működés biztosítja.

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

I.2. A C/C++ analizátor front end

A nyelvek bonyolultsága miatt az elemzés nem végezhető el a hatékony elemzési technológiákat használva ($LL(1)$ vagy $LALR(1)$). Továbbá, egy teljes front end-nek meg kell oldania a felismert forrás megfelelő belső reprezentációját, valamint annak továbbadását az analizáló eszközök további részeinek. Ezen felül a fent áttekintett egyéb funkcionális és nem funkcionális követelményeknek is eleget kell tennie.

Az analízis teljességét és a minden részletre kiterjedő reprezentációt úgy oldottuk meg, hogy a front end-ünkben két eszközt használunk. Hasonlóan a C/C++ fordítóprogramokhoz, különálló előfeldolgozó (preprocessor) végzi el a neki szánt feladatokat, miközben kigyűjti a szükséges információkat a modell előállításához is. Az előfeldolgozási elemektől mentes forrás további feldolgozását a fő nyelvi elemző végzi, amely ugyancsak előállítja a megfelelő modellt. Az előfeldolgozó elvégzi a bemeneti forrásfájlok szabályos elemzését, előállítva az előfeldolgozott forrást (`.i` fájl, vagy belsőleg átadott adatfolyam), amely azonos a fordítók által előállított forrással. Emellett kiadja az előfeldolgozó sémának megfelelő modellt is, amelyet a további analízisekhez lehet felhasználni. A sémát a [21] munkában ismertettük, és az tartalmaz minden részletes információt a forrás alacsony szintű elemeiről, úgy mint az egyes tokenek és egyéb előfeldolgozási információk (például makrók és helyettesítéseik). Az előfeldolgozó hasonló technológiát használ a modelljének felépítéséhez, mint amit itt ismertetünk. A következő lépésben a nyelvi elemző dolgozza fel az előfeldolgozott forrást és állítja elő a megfelelő nyelvi forráskód-modellt. A két modell együttesen képviseli az eredeti forrásfájlból kigyűjtött összes információt. Jelen dolgozatban a C/C++ nyelvi elemzővel foglalkozunk, amely magában foglalja a szintaktikus elemzőt és a megfelelő modell előállítását végző mechanizmust.

Az elemző egy parancssori alkalmazás, amely elősegíti a legáltalánosabb felhasználását. A program bemenete egy teljes fordítási egység, előfeldolgozott forrás formájában, amit a szintaktikus elemző fogad és ellenőriz a nyelvi megfelelőség szempontjából. A forrásnak megfelelő belső reprezentáció elemzés közben készül, a szintaxis által vezérelve. Ez azt jelenti, hogy ahogy az elemző egy nyelvi elemet feldolgoz, közben ún. *szintaktikus akciókat* hajt végre. Az akciók gyakorlatilag egy interfészt képeznek, amely a szintaxisnak megfelelő operációkat definiálja. Az interfészt megvalósítva, az elemző által hívott operációk használhatók fel ezután a „hasznos munka" elvégzésére, esetünkben a belső reprezentáció és egyben a forráskód-modell elkészítésére. A modell gyakorlatilag egy Absztrakt Szintaxis Gráf (ASG), amely megfelel a keretrendszerünk által definiált sémának, és az nem más, mint az absztrakt szintaxis fa megfelelője kiegészítve a csomópontok közötti különböző relációkkal. Ilyenek például a feloldott név-hivatkozások, de részletesebb analíziseket, például folyamgráf számítást már nem végzünk a gráfon. Az ASG szolgálja még az elemzést is, ugyanis a C++ nyelv korrekt elemzéséhez szükség van a szimbólumokról különböző strukturált információk eltárolására. Például az egyes azonosítók jelentését rögzíteni kell, amely a szintaktikus környezetből adódik (típus, változó, stb.), hiszen az elemzés szintaktikailag kétértelmű szituációit csak ezen információkat felhasználva lehet feloldani. Ezen információkat az éppen építés alatt lévő ASG-ből nyerjük ki. A modellt fájlba is kiírjuk, ami az alapvető kapcsolódási pontot fogja jelenteni az analizátorok további részeivel, valamint a feldolgozására rendelkezésre áll egy programozási interfész (API) is.

Az elemzőnk hagyományos felépítésű: a bemeneti szöveget először tokenizáljuk, azaz elvégezzük a lexikális elemzést, amely előállítja a szintaktikus elemző számára a token-folyamot, mint bemeneti szimbólumsorozatot. A lexikális elemzés viszonylag könnyen elvégezhető determinisztikus véges automatával, hiszen az erre vonatkozó nyelvi definíciót reguláris kifejezésekkel is megadhatjuk. A lexikális elemző végez el még néhány további alacsony szintű feldolgozást, mint például a kommentek, nem nyomtatható üres karakterek, stb. felismerését. Mivel a C++, nyelvi szintaxisát tekintve, nem könnyen elemezhető nyelv, az elemzéshez különféle megoldásokat használnak, de szinte egyik esetben sem tiszta elemzési stratégiát. Általában valamilyen kiegészítő, az alap-stratégiát segítő megoldásokkal teli, sokszor kézzel írott elemző a végeredmény. Sőt, a nyelv komplexitásának kezelésére egyes kutatók bonyolult és költséges technológiákat javasolnak, mint amilyenek az egyes, természetes nyelvek elemzésére kidolgozott módszerek.

Az elemzőnkben alkalmazott technológia felülről lefelé haladó (már a nyelv megalkotásakor e stratégiát javasolták, továbbá számos fordítóprogram is ezt használja), erősen $LL(k)$-alapú algoritmus, $k = 2$-vel. Az algoritmus rekurzív alászálló eljárásokon alapul, ahol az eljárások egy-egy szabályt valósítanak meg. Adott szabály jobb-oldalainak megfelel a hozzá tartozó eljárás törzse, ahol az alternatíva eldöntését az erősen $LL(k)$ feltétel biztosítja. A jobb-oldalon szereplő nemterminális az adott eljárás meghívását, a terminális pedig a bemenet léptetését és a következő szimbólum beolvasását jelenti, a szimbólumok sorrendjének megfelelően (bal levezetés). Az elemzőt a nyelvtani leírásból automatikusan generáltatjuk a PCCTS rendszer segítségével [18], amely előállítja mind a lexikális, mind a szintaktikus elemző C++ forráskódját. A PCCTS támogatja az ún. *predikátumos* $LL(k)$ elemzést, amely az elemzési nehézségek feloldására rendkívül hatékony módszer (ld. [17]). A módszer alapja az ún. *szintaktikus* és *szemantikus predikátumok* használata, amelyek eldöntik az elemzés további irányát az olyan alternatíváknál, ahol az erősen $LL(k)$ feltétel nem teljesül és emiatt kétértelmű lenne a nyelvtan. Szintaktikus predikátumok segítségével gyakorlatilag korlátlan hosszúságú előrenézés, a szemantikus predikátumokkal pedig a bemeneten következő valahány szimbólumra vonatkozó szemantikus információ révén oldjuk fel a kétértelműségeket.

A forráskód-modell (ASG) építéséhez a már említett szintaktikus akció interfészt használjuk fel. Az alkalmazott rekurzív alászálló eljárásokat használó felülről lefelé haladó elemzés az építést logikus módon vezérelheti. Ehhez azt a mechanizmust használjuk, hogy a rekurzív eljárások végrehajtása

közben minden lényeges ponton végrehajtunk egy akciót, amely implementációja végzi a tényleges építést. Akciókat tipikusan az alternatívák elején és végén, valamint az egyes nemterminálisoknak megfelelő eljáráshívások körül helyezünk el. Az ASG építésén felül, az akcióhívások végrehajtása az objektumorientáltságot kihasználva egy általános felhasználást tesz lehetővé. Az interfész az elemző még több lehetőséget kínáló, komponens szintű újrafelhasználását segíti elő egy specializált eszközben.

Az előállított modell a Columbus sémának felel meg, amely az elemzővel egyidejűleg fejlődve érte el a mai, stabilnak tekinthető formáját [5, 12]. Az építés során számos segédstruktúrát és kiegészítő algoritmust kell használni, hiszen az elemző felől érkező akciók hívási sorrendje rögzített, ami nem mindig használható fel az azonnali építésre. Továbbá, a nyelv rekurzívan beágyazható jellege miatt számos helyen kell különböző vermeket használni az egyes elkészült elemek tárolására. Viszonylag összetett megoldást igényelt még a nevek feloldása, azaz a megfelelő entitás megtalálása egy névvel történő hivatkozás esetén. Függvényhívások esetén például ez magában foglalja az argumentumok típusainak meghatározását a standard konverziók figyelembevételével, a konstansok propagációját és végül a túlterhelt függvények feloldását. Ezen kívül számos egyéb kiegészítő feldolgozást is el kell végeznie az építőnek, amelyekre itt külön nem térünk ki.

A fent áttekintett követelmények teljesítésére számos speciális megoldást alkalmaztunk a C/C++ front end-ben. A minél jobb hibatűrés elérése érdekében az elemzőbe beépítettünk egy, többnyire heurisztikákon alapuló mechanizmust, amely az esetleges elemzési szintaktikus hibák esetén lép életbe. Ilyenkor a hibaüzenet után az elemző megpróbál helyrejönni és egy alkalmas következő pontnál folytatni az elemzést. A megoldás lényege, hogy hiba esetén a bemeneten átugrunk néhány soron következő tokent, egy nagyobb nyelvi elemet képviselő tokennel bezárólag, mint például a deklarációkat és utasításokat lezáró pontosvessző. Ezzel egyidejűleg az elemző is az adott szabálynál folytatja az elemzést. Az analizálási sebesség javítására használható az ún. előfordított fejlécfájlok technikája, amikor a közös kódokat tartalmazó, sokszor használt fejlécfájlokat előre leelemezzük és az előállított modelleket felhasználjuk a további különálló fordítási egységek elemzésénél. A C++ elemző a standardnak megfelelő nyelvet ismeri fel, és a modellt is annak megfelelően építi fel. Az elemző ezen kívül támogatja még a Microsoft, a Borland és a GNU g++ fordítók dialektusait, továbbá lehetőség van a felismert nyelv bizonyos szintű bővítésére is azáltal, hogy egyes előredefiniált szimbólumokat megadunk az elemzőnek.

A front end tartalmaz még további lehetőségeket is, amelyekkel a programmegértést támogatjuk. Ilyen a C++ sablonok példányosítása forráskód szinten, amelyről a [11] cikkünkben található több információ.

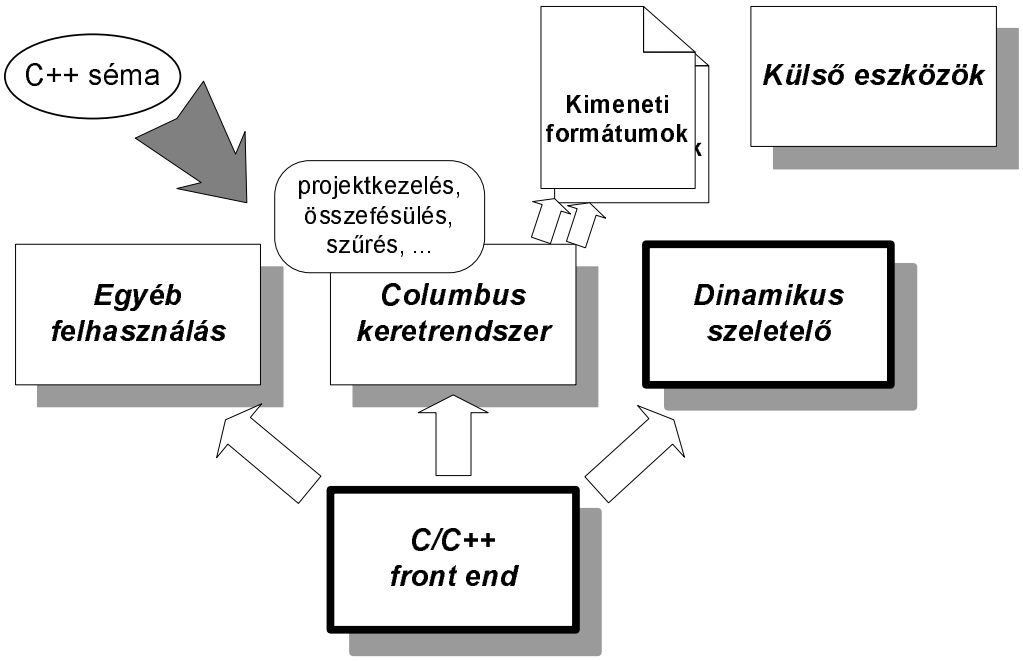
A C++ front end számos alkalmazásban bizonyította használhatóságát. Az analizálás sebessége, valamint a felhasznált memória kivételesen nagy rendszerek esetében is nagyon jónak mondható. A [10] cikkben közzétett eredmények alapján elmondhatjuk, hogy a memória-foglalás közel lineáris a rendszer méretével (pl. az osztályok számát tekintve). Továbbá, az analizálási idő a fordítási idővel összemérhető. Az említett kísérletben valós méretű, 5 000 osztályból álló rendszert is megvizsgáltunk.

Az analizátor front end-ünk minden további analízis alapja, beleértve azon egyszerűbb tényfeltárókat, amelyek például a kód formázott dokumentációját állítják elő, de a bonyolultabb számítást végző algoritmusokat is, mint amilyen az adatfolyam számítás. Egy teljes szoftverrendszer sikeres analíziséhez ugyanakkor számos egyéb dologra is szükség van. Számunkra ezeket a Columbus keretrendszer nyújtja [6, 7, 10], amelybe olyan fontos dolgok lettek beépítve mint a projektkezelés, valamint a modellek összefésülése és szűrése. A Columbus nincs kötve adott programozási nyelvhez sem, jelenleg a C++-hoz tartozó front end parancssori programként van behívva. A jelenlegi Columbus rendszer néhány hasznos feldolgozást is tartalmaz a C++ nyelvhez. Ilyenek a sémának megfelelő információkat tartalmazó különböző formátumok, pl. XML-alapú modellek, UML szerkezeti (osztály) modell és egyéb eszközök által definiált csere-formátumok. Ezen kívül a Columbus rendszer képes

előállítani még további származtatott eredményeket is, amelyek speciális célt szolgálnak, például metrikák számítása, tervezési minták felismerése és forráskód auditálás. A Columbus rendszer és annak kimeneti eredményei számos egyetemi együttműködésünk tárgya. Akadémiai felhasználóból szerte a világon eddig több mint 600-at regisztráltunk.

A Columbus rendszertől függetlenül is a front end-nek számtalan alkalmazására nyílik lehetőség: tulajdonképpen minden olyan területen, ahol C++ elemzésre van szükség. A szintaktikus akciók interfészét felhasználva az elemző további alkalmazására van lehetőség. Effajta bővítéssel eddig különböző *forráskód instrumentálási* alkalmazásoknál kísérleteztünk (itt az eredeti forrást kiegészítjük olyan utasításokkal, melyek további tevékenységet végeznek az eredeti viselkedés megtartása mellett). Instrumentálással érjük el például a dinamikus programszeletelés alkalmazásban a végrehajtási nyom előállítását. A dinamikus szeletelés a dolgozat második részének témája.

A front end helyét a programmegértést támogató környezetünkben az 1. ábra mutatja.



1. ábra. A C/C++ front end helye a programmegértést támogató környezetünkben.

I.3. I. rész eredményeinek tézisszerű összefoglalása

A C/C++ forráskód analízis és modell-építés kapcsán a következő eredményeket értem el:

I/1. *C/C++ forráskód analízis és modell-építés.* Kifejlesztettem a szükséges technológiákat és megvalósítottam egy analizátor front end-et, amely teljesíti az általános analizátoroknak támasztott követelményeket, emellett az eszköz egy jól meghatározott sémának megfelelő modellt képes előállítani, és könnyű bővíthetőséget lehetővé tevő interfészei vannak. Egyedi megoldásokat alkalmaztam többek között a hibatűrés, analizálási sebesség, dialektusok kezelésére.

Kapcsolódó publikációkban szereplő eredmények A [5] cikkben ismertettük a C++ sémát, amelynek megfelelő modellt épít fel a front end, sajátos technológiával (a séma nem, de az építés technológiája a szerző eredménye). A [10] publikációban mutattuk be az általános C++ front end-ekkel szemben támasztott követelményeket. Továbbá, ismertettük az analizátorunkat főbb tulajdonságaival és néhány mérési eredményt tettünk közzé, ezek a szerző munkái. A [21] közlésben ismertettük az előfeldolgozót és sémát, valamint néhány minta modellt (az előfeldolgozási séma nem a szerző munkája). A előfeldolgozóban használt építési stratégia és technológia a C++ nyelvi elemzőben alkalmazott alapján készült, amely a szerző eredménye.

II. Dinamikus programszeletelés

II.1. Programszeletelés

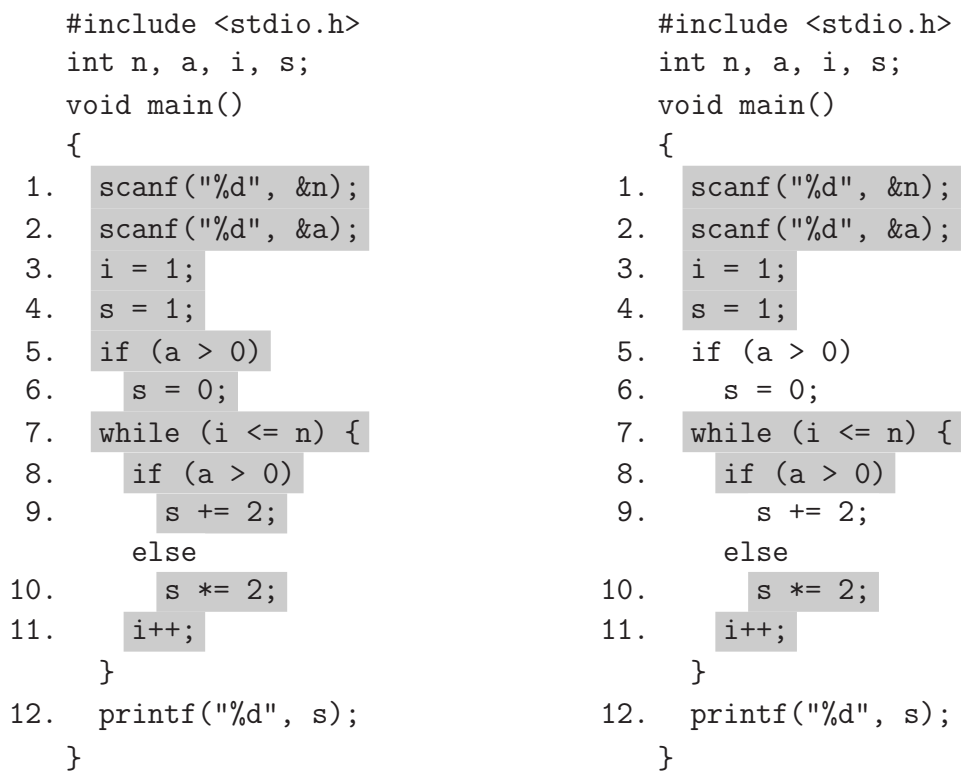
A *programszeletelés* (program slicing) és a szeletek fogalmát számos módon definiálták már az irodalomban, de az alapvető ötlet mindig ugyanaz: próbáljuk meg úgy csökkenteni a problémánk méretét, hogy az analizálandó programnak csak a vizsgálatunk szempontjából lényeges részét kelljen figyelembe vennünk. A szeletelés [19, 22] egyfajta programanalízis, amelyet számos szoftverfejlesztési és -karbantartási feladathoz használhatunk. Ide tartoznak például a program verifikáció, karbantartás, újratervezés, programmegértés és nyomkövetés. Általánosan szólva, a szelet a programnak egy olyan részhalmaza, amely tartalmazza azon utasításokat, melyek közvetlenül vagy közvetetten kihatással voltak vagy lehetnek egy adott program-pont adott változó-előfordulásainak értékeire (ezen előfordulások és program-pontok alkotják a szeletelési kritériumot). A részprogram meghatározásának folyamatát nevezzük szeletelésnek.[2] Ez azt jelenti, hogy az áttételes függőségeket is figyelembe véve, a csökkentett program a nevezett változókat azonosan fogja kiértékelni az adott előfordulásnál. Sőt, egyes szeletelési algoritmusok ténylegesen végrehajtható programot állítanak elő, míg mások tetszőleges részprogramot produkálhatnak. Ugyanakkor, a végrehajthatóságot csak néhány alkalmazás követeli meg, itt viszont számolni kell azzal a ténnyel, hogy az ilyen szeletek kevésbé precízek.

Ha a részprogramot úgy határozzuk meg, hogy az a program bármely futásakor fellépő viszonyokat magában foglalja, akkor *statikus szeletelésről* beszélünk, ha viszont csak egy konkrét futáshoz tartozó viszonyokat kell tartalmaznia, akkor az *dinamikus szeletelés*. A dinamikus szeletelési kritérium egy konkrét program-futás paramétereit is magában foglalja, így egy tesztesetet képezve (egy adott program bemenet), valamint a kérdéses utasítás egy konkrét előfordulását a program futása során (mivel ugyanazon utasítás többször is szerepelhet a végrehajtásban, a különböző előfordulásokat utasítás helyett *akciónak* fogjuk nevezni, amely magában foglalja az utasítás előfordulási sorszámát is). A tesztesethez tartozó programfutáskor végrehajtott akciók sorozatát és egyéb végrehajtási információkat az ún. *végrehajtási nyom* (execution trace) rögzíti. A 2. ábrán láthatunk egy példát a statikus és dinamikus szelet közötti különbségre (a statikus szelet az egész programot tartalmazza).

Az évek során számos szeletelési módszer került kidolgozásra. A gyakorlati módszerek többsége a program elemei (változók, utasítások, címek, predikátumok, stb.) között fellépő különböző függőségek (úgy mint adatfüggések és vezérlési függések) alapján számolja ki a szeleteket. A statikus szeletelési módszerek mélyreható részletességgel vannak megadva az irodalomban. Például Horwitz *és mások* [14] munkája számos későbbi implementáció és finomítás alapja lett, amelynek az alapja az ún. program-függőségi gráf – PDG. Ugyanakkor viszonylag kevés publikáció jelent meg, amely a dinamikus szeletelést a gyakorlati oldaláról közelíti meg, és részletes algoritmust ad meg. Az, hogy a gyakorlatban is használt algoritmusokkal nem igazán találkozhatunk, jelentheti azt is, hogy a közölt módszerek nem alkalmasak valós méretű programok kezelésére.

A meglévő algoritmusok dinamikus szeletek meghatározására különböző megközelítéseket használnak. Az első dinamikus szeletelési algoritmust Korel és Laski publikálta, amely végrehajtható szeleteket számolt ki [16]. Mint tudjuk, a végrehajtható szeletek általában lényegesen nagyobbak, mint a pusztán függőségeket figyelembe vevő módszerek (Venkatesh mérései szerint az arány körülbelül 2–3-szoros [20]), ugyanakkor a mi munkánkban nem követeltük meg e tulajdonságot, így lényegesen pontosabb szeleteket kapva. Az egyik legjelentősebb dinamikus szeletelési algoritmust Agrawal és Horgan fejlesztették ki [1], amelynek néhány továbbfejlesztett változatával is találkozhatunk [15, 23]. Az alap módszer az akciók közötti függőségek vizsgálatán alapul, amelyeket az ún. *dinamikus függőségi gráf*

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



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

(DDG) segítségével reprezentálják. A gráf egy csomópontja egy akciónak felel meg, a közöttük lévő élek pedig az akciók közötti dinamikus adat- és vezérlési függőségeket képviselik. A DDG alapján már könnyű a dinamikus szelet meghatározása: a dinamikus szeletelési kritériumban szereplő akcióból kiindulva az élek mentén haladva bejárjuk a gráfot, és a közben érintett akciók utasításai fogják képezni a dinamikus szeletet. Az eredeti módszer legnagyobb hátránya az, hogy az így kapott gráf rendkívül nagy lehet, hiszen annyi csomópontja lesz, ahány akció szerepel a végrehajtási nyomban, az élek számát pedig a dinamikus függőségek határozzák meg, aminek eredménye a gráf gyakorlatilag kezelhetetlen mérete általános esetben. Ezen problémán felül, a publikált módszerek nem ismertetnek elegendő részletet valós programozási nyelv analízisére vonatkozóan, nem térnek ki számos jelentős probléma kezelésére.

II.2. Hatékony dinamikus szeletelési módszereink

Az általunk kidolgozott dinamikus szeletelési algoritmusok lényegesen különböznek a már meglévő módszerektől. Algoritmusaink hatékonyan implementálhatóak és ezáltal a gyakorlatban is használhatók valós programok analíziséhez. Ehhez az is hozzájárul, hogy a számításokhoz szükséges tárolt adatok mennyiségének csökkentésére törekedtünk azáltal, hogy egyszerű adatszerkezeteket definiálunk. Két, dinamikus függőségeken alapuló algoritmust adunk meg, melyek közül az első *globális*, azaz egy adott futáshoz az összes dinamikus szeletet kiszámolja, azaz nem tartozik hozzá egy konkrét szeletelési kritérium (a globális algoritmust először a [13] cikkben publikáltuk, amit követett a C-re való kiegészítés a [3] publikációban). A második módszer *igényvezérelt*, ami azt jelenti, hogy egyetlen szeletet határoz meg egy kérés (szeletelési kritérium) alapján. Mindkét algoritmus ugyanazon szeleteket képes meghatározni, mint a hagyományos dinamikus függőségi gráfon alapuló módszer, de lényeges, hogy merőben más megközelítést alkalmazunk. Mindkettőnek megvannak az előnyei és hátrányai, ezért a módszerek alkalmazási körei is különbözőek. A módszerek alapelvét először röviden ismertetjük, majd megadjuk a szükséges kiegészítéseket C programok szeletelésére.

Módszereink alapját a program-elemek közötti függőségek számítása képezi. Mindkét módszernél az utasítás-előfordulások (akciók) között dinamikusan fellépő adat- és vezérlési (kontroll) függőségeket követjük. A statikus függőség-alapú módszerek esetében minden lehetséges függőséget figyelembe kell venni (lásd pl. Horwitz *és mások* munkáját [14]), és e célból meg kell konstruálni a program-függőségi gráfot (PDG). Ezzel szemben, dinamikus esetben, amikor egy konkrét futást vizsgálunk, csak a ténylegesen megvalósult (dinamikus) függőségeket kell figyelembe venni. Akciók közötti függőség vezérlési esetben a legutóbb végrehajtott akciót jelenti azok közül, amelyek az adott függő utasításra (statikus értelemben véve) potenciálisan kihatnak. A megvalósult adatfüggéseket pedig a változók utolsó definiálási program-pontjuk alapján határozzuk meg. Ezek alapján nincs szükségünk teljes statikus vagy dinamikus függőségi gráfra. Ehelyett sokkal egyszerűbb statikus struktúra kiszámítására van szükség, amelyet később mindkét algoritmus fel fog használni a szelet kiszámítása során.

Az algoritmusok közös statikus analizálási fázissal rendelkeznek, melynek során előállítjuk az ún. D/U programreprezentációt (definiálás-használat). A program egy utasításának D/U reprezentációja a következő formájú: $i.\ d:\ U$, ahol az i-edik utasításban értéket kapó (definiált) változó a d, továbbá U a d kiszámításához i-ben felhasznált változók halmaza (használati halmaz). A D/U reprezentáció az adatfüggéseket oly módon rögzíti, hogy csak a változók definiálásának és használatának tényét veszi figyelembe utasításonként, és nem tárolja a változó előfordulások közötti konkrét kapcsolatokat (más szóval, csak a változók nevei az érdekesek és nem az előfordulások). Továbbá, segítségével a vezérlési függőségeket is azonos módon rögzíthetjük mint az adatfüggőségeket. Ezáltal a szeletelési algoritmusok is egyszerűbbek lehetnek, mivel a kétféle függőséget azonosan kezelhetik. A hagyományos módszereknél nem használtak ilyen általánosítást. A vezérlési függőségeket az ún. *predikátum változók* képviselik, melyek az utasítások közötti közvetlen vezérlési függőségeknek megfelelő virtuális változók. Ezután a D/U reprezentációt felhasználva a két algoritmus a végrehajtási nyom feldolgozásával számítja ki a megfelelő szelet(ek)et. Egyszerűsége folytán ezen adatszerkezet tárolása és használata nem von maga után jelentős többletköltségeket.

A globális algoritmus a nyomot az első végrehajtott utasítással kezdve olvassa be, és minden lépésnél meghatározza az aktuális utasításban definiált változóhoz tartozó függőségi halmazt, amihez a használt változók legutóbb kiszámolt függőségi halmazait és a legutóbbi definiáló utasításait használja fel (ld. [3, 13] cikkeket). Ily módon minden dinamikus szeletet megkapunk és kimenetre adunk a definiált változókhoz az utasítások minden előfordulásánál (a memóriában csak az érvényben lévő halmazokat tároljuk). Ez a módszer ott alkalmas, ahol több szelet kiszámítására is szükség van a végrehajtási nyom egyszeri feldolgozásával. Az algoritmus iterációinak száma egyenlő a végrehajtás hosszával, ezen felül az átlagos műveletigényét a függőségi halmazok átlagos mérete határozza meg, amely általában a program méretével van összefüggésben (ez gyakorlatilag a szeletek mérete). A tárigényt a halmazok számának és méretének szorzata határozza meg: az első értéke az összes definiált változó száma, míg a második az átlagos szelet-méret. Látható, hogy a tárigény lényegesen jobb mint a gráf-alapú módszernél. Az algoritmus a 3. ábrán látható. Itt, $DynDep$ jelöli a megfelelő függőségi halmazokat, LS pedig a legutóbbi definiáló utasítást. Továbbá, i^j egy akció az utasítás-sorszámmal és a végrehajtási lépéssel, valamint d a megfelelő utasításban definiált változó, U pedig a használati halmaz. A többi részlet az értekezésben megtalálható.

Az igényvezérelt algoritmus minden egyes igény esetén a nyomot újra feldolgozza a kritériumban szereplő akcióval kezdve, majd hátrafelé haladva az első akció felé követi vissza a függőségeket. Az algoritmus bejárja a dinamikus függőségeket és a még fel nem dolgozott akciókat egy munkahalmazba gyűjti. A munkahalmazból egy elemet kivéve megnézi annak használati halmazában szereplő összes változó legutolsó definíciójához tartozó akciót az aktuális lépés előtt, és ezekkel bővíti a munkhalmazt. Amikor minden függőség fel lett oldva és a munkahalmaz kiürül, akkor fejezi be a számítást és adja vissza szeletként a közben érintett utasításokat. Az algoritmus működéséhez a végrehajtási

program GlobalisAlgoritmus($P, \mathbf{x}$)
input: P : program
$\mathbf{x}$: program bemenet
output: dinamikus szeletek minden $(\mathbf{x}, i^j, U(i))$ kritériumhoz

begin
  Végrehajtási nyom rögzítése
  for $j = 1$ **to** lépések száma
    $i :=$ j-edik lépésben végrehajtott utasítás
    $DynDep(d(i)) := \bigcup_{u_k \in U(i)}(DynDep(u_k) \cup \{LS(u_k)\})$
    $LS(d(i)) := i$
    $DynDep(d(i))$ kimenetre, mint $(\mathbf{x}, i^j, U(i))$ kritérium dinamikus szelete
  endfor
end

3. ábra. Globális algoritmus

nyomot egy speciális formában kell tárolni, amely az egyes akciókat aszerint rendeli össze, hogy az adott utasításban mely változó lett definiálva. Ennek reprezentálására az ún. EHT táblát használjuk, melynek soraiban fel vannak sorolva a megfelelő definiált változókhoz tartozó akciók. Erre azért van szükség, mert az algoritmus egy adott iterációjában valamely változó definíciói közül tetszőlegesre szükség lehet, tehát a végrehajtási lépések között visszamenőleg keresni kell. Az algoritmus iterációinak száma változó, legalább a kiszámolt szelet méretével megegyező, de legrosszabb esetben a végrehajtás hossza is lehet. Ugyanakkor, átlagos esetben a szelet méretével lesz arányban. A műveletigényt ezen felül az EHT táblában való keresés határozza meg, amely logaritmikus a lépésszám függvényében. Ami a tárigényt illeti, eltekintve a táblától (amely lemezen is tárolható), nem jelentős, hiszen csak a munkahalmazt kell a memóriában tartani.

A két algoritmust összehasonlítva elmondható, hogy a globális algoritmust akkor használjuk, amikor több szelet meghatározására van szükség egyszerre, például uniós vagy dekompozíciós szeletek számítására. Igény szerinti, egy dinamikus szeletelési kritériumhoz tartozó szelet meghatározásához mindkét módszer használható. Ha a végrehajtási történet teljes egészében történő tárolása nem célszerű, a globális algoritmust választhatjuk. Egyedi feladatokhoz általában az igényvezérelt algoritmus a jobb választás kevesebb művelete és kisebb tárigénye miatt. Ugyanakkor, egyes hosszú végrehajtási történetekhez tartozó igény szerinti szelet meghatározására csak akkor alkalmazható hatékonyan ez a módszer, ha az EHT tábla tárolását és a benne történő keresést hatékonyan tudjuk megoldani. Végül, ha sok szeletet akarunk kiszámítani az igényvezérelt algoritmust egyenként lefuttatva, akkor az összköltség a szeletek számának egy határa fölött már nagyobb lesz a globális algoritmusénál.

A módszerek alkalmazásához C nyelvű programok szeletelésére számos egyedi problémát kellett megoldani és az algoritmusokat kiegészíteni, de ezek nem befolyásolták lényegesen a komplexitást. Az általunk alkalmazott megvalósításban a végrehajtási nyomot az eredeti program instrumentált változatának futtatásával állítjuk elő, amely előállítja a futásról szükséges minden információt. Az instrumentálást és a statikus analízist a C/C++ front end-ünkkel végezzük a fent vázolt interfészek felhasználásával. Vagyis, a kiegészített D/U reprezentációt az ASG-ből származtatjuk, míg az akció interfészt felhasználva hozzuk létre az instrumentált kódot (úgy, hogy elérjük a felismert tokeneket és azokat visszaírjuk az instrumentáló utasításokkal kiegészítve).

Az alábbi pontokban összefoglaljuk a legfontosabb kiegészítéseket C szeleteléséhez:

- A legfontosabb talán az, hogy a skaláris változókat, mutatókat és más összetett objektumot

egységesen kezeljük azáltal, hogy minden számítást *memória cellákon* végzünk. E megközelítéssel rendkívül leegyszerűsödik az említett programelemek kezelése, hiszen – ellentétben a statikus szeleteléssel, ahol minden eshetőséget vizsgálni kell – itt minden dinamikus információ rendelkezésre áll az aktuális programfutás objektumainak tárolásáról. Először mindent át-transzformálunk memóriacímekké: a skalárisoknak az aktuálisan felvett címeit, valamint a mutatók, tömb- és struktúra-elemek konkrét címértékeit használjuk, továbbá a szeletelési kritériumban szereplő változókat is címmé alakítjuk. Ezután az algoritmusok ezen címekkel dolgoznak, mint „változókkal".

- A D/U programreprezentációt is számos szempontból ki kellett egészíteni, mely kiegészítések viszont nem befolyásolják a komplexitást. Egy ilyen kiegészítés az, hogy néhány további virtuális változót vezetünk be a mutató indirekciók, struktúra-mezők elérésére, valamint a függvényhívások kezelésére. A mutatók és egyéb indirekciók (tömb-elemek, mezők) kezelésére bevezetjük az ún. *dereferencia változókat*, melyeket a statikus D/U-ban szimbolikusan használunk. Ezeket a végrehajtás feldolgozásakor valódi címekkel helyettesítjük, melyeket a nyomból olvasunk ki.

- Jelentős kiegészítés a vezérlési függőségek kezelése, ugyanis C-ben a tetszőleges, nemstrukturált vezérlésátadások (pl. *goto* utasítás) miatt teljes vezérlési függőségi gráfot kell építeni, majd ez alapján elkészíteni a D/U reprezentáció predikátum változóit. Általános esetben egy utasítás több predikátumtól is függhet (statikus értelemben véve), egy konkrét futáskor azonban csak egy függőség realizálódik. A szeletelő algoritmusok ezt úgy kezelik, hogy a potenciális függőségek közül a legutoljára végrehajtottat tekintik megvalósult függőségnek.

- További részletek még például a könyvtári függvények kezelése előre elkészített D/U struktúrák felhasználásával, több fordítási egység közös kezelése, valamint a fizikai programsor-számok és logikai utasítás-sorszámok összerendelése.

A fenti kiegészítéseken túl az igényvezérelt algoritmus érdemel némi további figyelmet, itt ugyanis szükség van még egy további segéd-struktúra karbantartására. Ez az ún. AHT tábla, amely a skaláris változók és dereferenciák által felvett címek történetét tárolja egy kompakt formában. A végrehajtás feldolgozása közben kikereshető belőle tetszőleges változó által felvett cím egy tetszőleges lépésnél.

Az algoritmusok komplexitását befolyásoló legfontosabb tényező az, hogy a függőségeket memória cellákon számoljuk, tehát az összes lehetséges változó száma csak futás közben lesz ismert, és az a használt különböző cellák számától függ. Ez azt eredményezi, hogy a megvalósítás egyes helyein költségesebb adatszerkezeteket kell alkalmazni. Ugyanakkor méréseink azt igazolják, hogy a különböző címek száma általában nem jelentős a programmérethez és a statikus változószámhoz képest és nem jellemző a futás hosszától való függés sem, ezért ezek nem jelentenek kezelhetetlen többletköltséget.

A megvalósított algoritmusokkal kiterjedt méréseket végeztünk el, melyek eredményei alapján beigazolódtak a komplexitással kapcsolatos elméleti vizsgálódásaink. Nevezetesen az, hogy a legrosszabb eset komplexitásánál lényegesen jobbak a valós tesztesetek művelet- és tárigényei, a mért értékek több nagyságrenddel jobbak. Valamint az, hogy mindkét módszer esetében a legfontosabb tényezők vagy a program mérettel, vagy pedig a használt különböző memória címek számával vannak összefüggésben, és nem pedig a végrehajtott lépések számával. A műveletigény és tárterület hatékonyságot nem közvetlenül a futási sebesség és memóriafoglalás mérésével vizsgáltuk, hiszen a jelenlegi prototípus, optimalizálatlan implementációk nem képviselnek reális eredményt. Ehelyett az algoritmusok futása közbeni számítások különböző paramétereit mértük (belső adatszerkezetek, lépésszámok), speciális kiíró rutinok beépítésével a megvalósításba. Méréseinkhez öt kis és közepes méretű programot használtunk fel. A mérések kiterjedtek a programreprezentáció méretére, szeletek méreteire és a két algoritmus számítási és tárterület komplexitására. Bebizonyosodott, hogy az igényvezérelt algoritmus általában gyorsabb, ha egy szelet kiszámítását nézzük, és ha nem tekintjük a táblák építését.

II.3. Dinamikus szeletelés alkalmazásai

A dinamikus szeletek legfontosabb alkalmazása a nyomkövetésnél van. Ehhez megadtunk egy kiegészítést, amely az ún. *releváns szeleteket* határozza meg, amelyek megbízhatóbb eredményt szolgáltatnak egyes kód-szerkezetekre, mint a hagyományos dinamikus szelet [13]. A probléma ott jelentkezik, hogy a dinamikus szelet egyes esetekben nem foglal magában olyan utasításokat, amelyek ugyan nem befolyásolták a kritérium változóit, de más kiértékelés mellet már befolyásolták volna azokat (azt mondjuk, hogy a változók *potenciálisan* függnek az adott utasításoktól). Itt már némi statikus függőségi információval is számolni kell, ugyanis az utasítások összes lehetséges kiértékelésének figyelembe vétele már statikus feldolgozás.

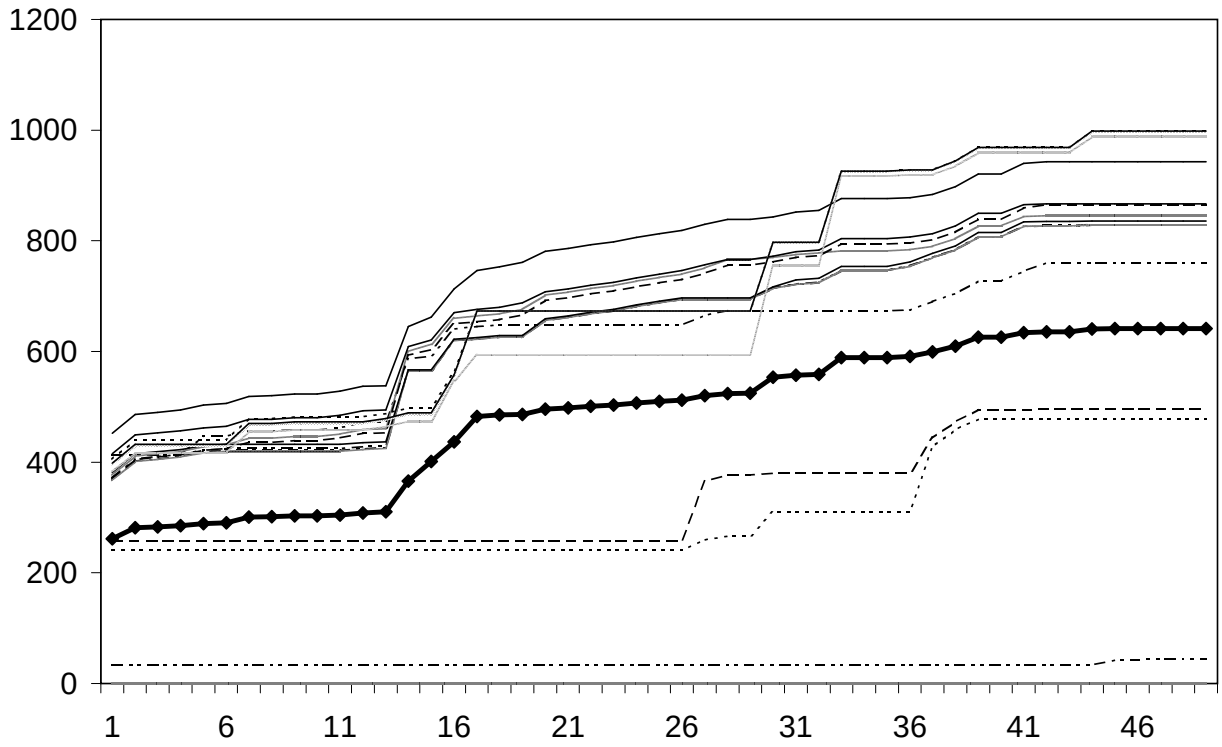
Azonban a dinamikus szeletelés szélesebb körben is alkalmazható, például a globális algoritmusunkat használva sok szelet határozható meg hatékonyan, és emiatt számos karbantartási és tesztelési feladat támogatható segítségével. Egy konkrét módszert is megadtunk, amellyel az ún. *uniós szeleteket* határozhatjuk meg, ahol a a program különböző végrehajtásaihoz tartozó dinamikus szeletek egyesítését számoljuk ki [2]. Az uniós szelettel az ún. *realizálható szeletet* közelítjük, amely csak elméletben létezik, és az összes lehetséges futáshoz tartozó dinamikus szelet unióját képviseli. A realizálható szelet kisebb a statikus szeletnél, viszont gyakorlati úton csak közelíteni tudjuk az uniós szeletekkel több tesztestre való futtatással. Az uniós szelet nagyobb mint bármely dinamikus szelet, viszont lényegesen kisebb a statikus szeletnél, tehát pontosabb is annál. Ezt az alapján állítjuk, hogy méréseinkkel azt tapasztaltuk, hogy a realizálható és a statikus szelet között lényeges lehet a különbség. Kísérleteink eredményei azt mutatják, hogy a dinamikus szeletek általában kicsik (a program megközelítőleg 5%-a), míg a statikus szeletek lényegesen nagyobbak (általában több mint a program 70%-át teszik ki). Ugyanakkor a már kevés számú, reprezentatív végrehajtással kiszámított uniós szelet növekedési sebessége is lényegesen lecsökken, és a tendencia változatlannak tűnik, amiből arra következtetünk, hogy a realizálható szeletet elég jól megközelítettük (ld. a 4. ábrát). Eközben a mérete jelentősen kisebb lesz a statikus szeletnél, méréseink szerint a program hozzávetőlegesen 15%-a. Ennek az lesz a haszna, hogy az uniós szeleteket számos statikus szeleten alapuló alkalmazásban használhatjuk fel utóbbiak helyett. Például a karbantartásnál valamilyen probléma megoldásához egy jól megválasztott teszteset-halmazzal kiszámított uniós szelettel kezdhetjük a vizsgálódásunkat, és mivel kisebb rész-programmal dolgozunk, valószínűleg hatékonyabb is lehet e tevékenységünk.

II.4. II. rész eredményeinek tézisszerű összefoglalása

A dinamikus programszeletelés témakörében a következő eredményeket értem el:

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

Kapcsolódó publikációkban szereplő eredmények A [13] cikkben jelentettük meg először a globális dinamikus szeletelő alap-algoritmust, amely nem a szerző eredménye. A részletek kidolgozása, a prototípus megtervezése, a mérések elvégzése a szerző munkája. A [3] publikáció ismerteti a C programok szeletelésére vonatkozó részleteket, amelyek kidolgozása a szerző munkája. A cikk elnyerte a „European Conference on Software Maintenance" konferencia legjobb



4. ábra. Uniós szeletek növekedése az egyik példaprogramhoz. A vízszintes tengely a tesztesetek hozzáadását jelenti, a függőleges az uniós szelet méretét utasításszámban. A görbék a különböző kritériumokhoz tartoznak, a vastag vonal azok átlaga.

munkájáért járó díjat, amely a legnagyobb európai és világviszonylatban is az egyik legjelentősebb szoftverkarbantartási konferencia.

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

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

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

Kapcsolódó publikációkban szereplő eredmények A releváns szeletek számítását a [13] cikkben tettük közzé, ahol az alap-algoritmus nem a szerző eredménye. Az algoritmus részleteinek kidolgozása, a prototípus megtervezése, a mérések elvégzése a szerző munkája. A [2] publikációban ismertettük az uniós szeletek motivációját és számításuk technológiáját, amely a szerző eredménye, egyetemben a mérések megtervezésével és kiértékelésével.

Irodalomjegyzék

[1] 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.

[2] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–21. IEEE Computer Society, October 2002.

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

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

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

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

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

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

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

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

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

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

[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] 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.

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

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

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

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

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

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

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

[23] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, Washington, D. C., June 2004.