Szoftver karbantartási módszerek preprocesszált nyelvi környezetben

Doktori értekezés tézisei

Vidács László

Témavezető:
Dr. Gyimóthy Tibor

Informatika Doktori Iskola
Informatikai Tanszékcsoport
Szegedi Tudományegyetem

Szeged 2009

Bevezető

Sokan tettek erőfeszítéseket a szoftverfejlesztés hatékonnyá tételére. A legismertebb módszerek és módszertanok a szoftver életciklus fejlesztési fázisát támogatják. A megnövekedett fejlesztési hatékonyság és a folyamatosan változó technológiák előtérbe hozták a szoftver karbantartás fontosságát. Hatalmas C és C++ rendszerek léteznek napjainkban, melyek már évek óta karbantartási fázisban vannak. Természeténél fogva a karbantartás más jellegű tevékenységeket követel meg, mint az megszokott a fejlesztés során. A szoftver karbantartás nem pusztán az előkerült hibák javítását jelenti. Ide tartozik minden a szoftver átadását követő módosítás, mely a rendszer teljesítményét vagy más mutatóit javítja, esetleg a megváltozott körülményekhez való igazodást szolgálja. Ennek költségeit nagyon könnyű alábecsülni. A hibák javítása és a szoftver korszerűsítése a változó körülményeknek vagy új felhasználói követelményeknek megfelelően igen költséges feladat. Egy működésben levő rendszeren minden változtatás sokkal költségesebb, mintha ugyanazt a munkát egy korábbi fázisban végeznénk el. Mindezek kiemelik a karbantartás támogatásának fontosságát. A program-megértés a karbantartási feladatok kritikus része. A fejlesztőnek, aki egy már elkészült rendszeren hajt végre módosítást, mind magas szintű ismeretekkel, mind implementációs szintű tudással rendelkeznie kell az adott rendszerről. Ezen információk kinyerése a visszatervezési (reverse engineering) folyamat [5]. A forráskód alapú visszatervezési módszerek (amilyen a mienk is) fontosságát növeli, hogy karbantartáskor az egyetlen megbízható dokumentáció maga a forráskód. A többi specifikáció, terv, dokumentáció hiányossá és pontatlanná válik az évek során. Továbbá egy elavult leírás félrevezető is lehet, ami újabb költségeket indukál.

Habár a fejlesztés és a karbantartás sokban hasonlítanak, a gyors fejlesztés és a jó karbantarthatóság sokszor ellentétes fogalmak. Ez a jelenség megfigyelhető a preprocesszor esetében is. A preprocesszor hasznos mivoltát a fejlesztésben való sokéves használata is bizonyítja. Kiterjedt használatát indokolja a rugalmas konfiguráció-kezelés, a forrásfájlok hierarchiába rendezése include-ok segítségével, illetve a szöveg-alapú makrók hasznossága. Egy gyakorlati tanulmány, melyet jól ismert unix programokon végeztek, kimutatta, hogy a direktívák a programsorok átlagosan 8.4%-át teszik ki [6]. A preprocesszor általános megítélése azonnal megváltozik amint karbantartásról, program-megértésről van szó: a direktívák jelenlétét mindig akadályként említik [17]. A makrók és feltételes direktívák egy része C/C++ kóddá alakítható [13], emellett módszereket dolgoztak ki a felesleges feltételes direktívák eltávolítására is [1], de a direktívák túlnyomó többsége a forráskódban marad. Az alapvető probléma a preprocesszorral az, hogy a fordító a feldolgozott forráskódot kapja meg, és nem az eredeti forráskódot amit a fejlesztő lát. Sok esetben a két forráskód nagy mértékben eltér. Ezek az eltérések megnehezítik a program megértését a fejlesztők számára, emellett gondot okoznak a program megértést támogató eszközöknél is. A visszatervező eszközöket gyakran használják amikor a karbantartónak nincs kellő ismerete a rendszerről. Az eszköztámogatásra még nagyobb szükség van preprocesszált nyelvek esetén, amikor a karbantartó csak az eredeti forráskódot figyelheti meg.

Munkánk célja a preprocesszor direktívák jelenléte miatt akadályozott szoftver karbantartási lépések támogatása. Sajnos a C/C++ elemzőeszközök a preprocesszorral kapcsolatos problémákat sok esetben teljesen figyelmen kívül hagyják, vagy csak felületesen kezelik, néhány üdítő kivételtől eltekintve [7, 12]. Munkánk központi eleme a preprocesszor részletes metamodellje (visszatervezési környezetben séma). A séma a forráskód elemeit írja le preprocesszálási szempontból, nemcsak a direktívák szerkezetét, de a preprocesszálás folyamatát is modellezi. Séma példányon egy gráfot értünk, mely egy konkrét C/C++ programhoz tartozik. A séma példány a visszatervezési folyamat eredménye, szerkezete a sémának felel meg. A példányban kinyert információk további program-megértési célokra használhatók, például a makró felfedés/elrejtés, vagy az include hierarchia elemzése. További téziseinknél is intenzíven használjuk a sémát.

Makrók modell szintű újraszervezése (refactoring) kapcsán kidolgoztuk szempontok egy listá-

ját mely alapján konkrét makróval kapcsolatos újraszervezési műveletek készíthetőek magasszintű újraszervezési sémák alapján. Eszköz architektúrát terveztünk makrók újraszervezésének tervezéséhez, végrehajtásához és a végső modellek ellenőrzéséhez.

A változott kód hatásának analízise olyan folyamat, mely egy nagy rendszerben történt változás hatásait igyekszik feltárni [3, 16]. Újító módszereket dolgoztunk ki a program szeletelés területén, mely egy alkalmas eszköz a változás hatásanalízisére is. A makrókkal kapcsolatos elemzéseket két lépésben integráltuk a programszeleteléssel. Először megalkottuk a Makró Függőségi Gráfot (Macro Dependence Graph – MDG), és definiáltuk rajta az előrehaladó és hátrahaladó makró szeleteket. A makró szeletek számítása olyan pontokra is kiterjed egy programban, amelyeket a hagyományos C/C++ szeletelés nem vesz figyelembe. Habár a Makró Függőségi Gráf igazi előnyeit csak a második lépésben használjuk ki, ahol a C/C++ nyelvi és makró szeleteket összekapcsoljuk. A két típusú (C/C++ és makró) szeletek mind előrehaladó, mind hátrahaladó esetben összekapcsolhatók. Megadtuk az összekapcsolt függőségi gráf és az összekapcsolt szeletek definícióit, emellett algoritmusokat is a szeletek számításához. A kidolgozott szeletelési módszereket a gyakorlatban is megvalósítottuk és kísérletek útján kiértékeltük valós programokon.

Eredményeinket öt tézisben foglaltuk össze, melyeket két részre osztottunk a kutatási témák alapján. A tézisfüzet további része a következő tézispontokat részletesen is bemutatja:

I/1 Metamodell a C/C++ preprocesszor nyelvhez

I/2 Makrók modell szintű újraszervezése

II/1 Makró szeletelés

II/2 C/C++ nyelvű és preprocesszor szeletek összekapcsolása

II/3 Szeletelési módszerek gyakorlati kiértékelése

I Preprocesszor direktívák modellezése és újraszervezése

Az első rész eredményei a forráskódból kinyert preprocesszor modellekel kapcsolatosak. Először a preprocesszor sémát mutatjuk be, mely alapul szolgál a további karabntartást támogató módszereknek. Ezután a legfrisebb munkánkról, a preprocesszor direktívák modell szintű, gráf transzformációs eszközökkel történő újraszervezéséről számolunk be.

I/1 Metamodell a C/C++ preprocesszor nyelvhez

Columbus séma a C/C++ preprocesszor nyelvhez

Első hozzájárulásunk a preprocessor séma (metamodel), mely kulcsfontosságú a visszatervezési folyamatban. A séma a C/C++ forrásfájlok minden preprocesszorhoz kapcsolódó elemét modellezi, emellett információt szolgáltat a preprocesszor műveleteiről is, mint például a makró kifejtés. Tudomásunk szerint ez az első nyilvános, általános célú séma a preprocesszorhoz. A séma entitásokból, azok attribútumaiból és a köztük lévő kapcsolatokból áll, így az UML osztálydiagram jelölést követve adjuk közre. Séma példányon (modell) egy gráfot értünk, mely egy adott C/C++ programhoz tartozik és tartalmaz minden preprocesszorral kapcsolatos konkrét információt.

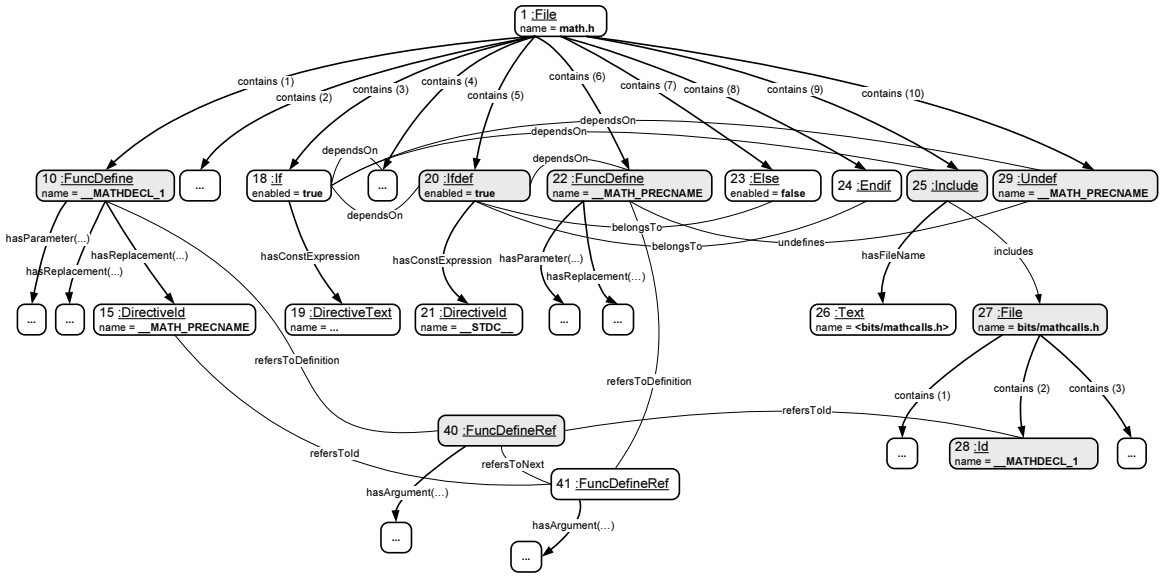
A séma példányokból mind az eredeti forráskód, mind a preprocesszált forráskód, valamint a preprocesszálási folyamat összes köztes állapota kinyerhető. Továbbá a séma leírja mind a dinamikus (konfigurációfüggő) és a statikus (konfigurációtól független) példányokat is. A bemutatott megoldás tehát alkalmas a preprocesszor használat részletes és teljes körű elemzésére.

A metamodell bemutatására itt nem térünk ki, de bemutatunk egy példa forráskód részletet (1. forráskód lista) és a hozzá tartozó séma példányt az 1. ábrán. Minden preprocesszor nyelvi elemet egy attribútumokkal ellátott csomópont reprezentál a gráfban, mint például a `__MATHDECL_1` makró definíciója (10), az `__STDC__` feltételes direktívája (20), a of `__MATH_PRECNAME` makró definíciója (22), egy include direktíva (25) és a hozzá tartozó forrásfájl részfája (27). Ezenkívül a makróhelyettesítés lépéseit is nyomon követhetjük a gráf élei segítségével (lásd a 40-es hivatkozó csomópontot).

```
#define __MATHDECL_1(type, function,suffix, args) \ /*ID=10*/
  extern type __MATH_PRECNAME(function,suffix) args __THROW
...
#if defined __USE_MISC || defined __USE_ISOC99
...
#ifdef __STDC__                                     /*ID=20*/
# define __MATH_PRECNAME(name,r) name##f##r         /*ID=22*/
#else
# define __MATH_PRECNAME(name,r) name/**/f/**/r
#endif
#include <bits/mathcalls.h>                         /*ID=25*/
#undef __MATH_PRECNAME
```

1. forráskód lista. Example code from `math.h`

Egy programozói API-t is készítettünk melyen keresztül a gráf felépítése és az információ kinyerése lehetséges. Az elemzőeszközök közötti kommunikációt és a program-megértést elősegítendő a séma példányokat GXL (Graph eXchange Language/Gráf Adatcserélő Nyelv [9]) és PPML (saját XML reprezentáció) formátumban is elérhetővé tesszük.

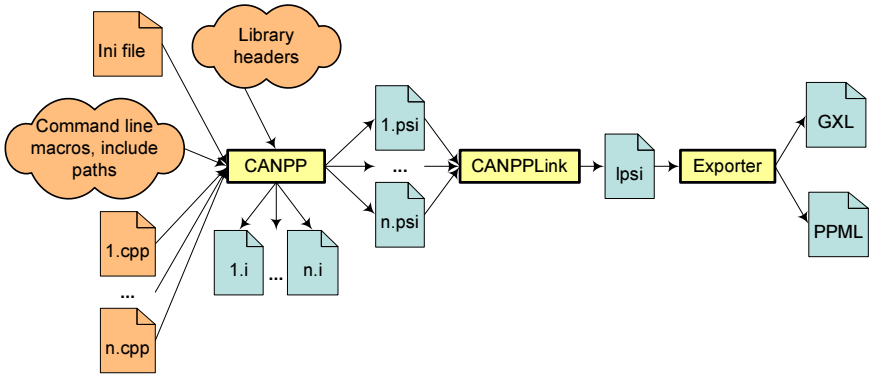


1. ábra. Dinamikus séma példány

Séma példányok építése

Implementáltunk egy preprocesszort, mely képes a séma példányok felépítésére az elemzett programok alapján. Az eszköz a Columbus keretrendszer részét képezi, és képes több millió programsoros, ipari méretű programok elemzésére. Egyelőre csak dinamikus séma példányok készítésére

alkalmas, de alkalmazásában így is számos lehetőség rejlik. Az eszköz a GNU gcc/cpp és a Microsoft cl preprocesszorok működését is követi. Emellett bizonyos szempontból hibatűrőbb is, például egy hiányzó header fájl nem állítja meg a teljes elemzést, ami visszatervezés esetén fontos tulajdonság. A visszatervezési folyamatot a 2. ábrán mutatjuk be. A forrásfájlokból a környezet beállításai alapján a hagyományos `.i` fájlok mellett elkészülnek a séma példányok, melyeket projekt szinten egybeszerkesztünk és végül szükség esetén XML formátumba exportálunk.



2. ábra. Preprocesszor direktívák visszatervezése

Eredmények hasznosulása

Egyrészt a többi tézis mindegyikének szerves részét képezi a kidolgozott séma. Másrészt a Columbus keretrendszeren belül is számos célra felhasználjuk a séma példányokat. Preprocesszoral kapcsolatos információval látjuk el a SourceAudit statikus kódolási szabályellenőrző eszközt. A kinyert include hierarchiát C/C++ programok inkrementális elemzésénél használjuk fel. Egy Visual Studio plugin-t kifejlesztve megvalósítottuk a makrók forráskódban történő lépésenkénti felfedését/elrejtését, [11] alapján.

Eredményeinket számos sikeres ipari és akadémiai kutatási projekt során felhasználtuk. Egy 5millió programsoros rendszert elemeztünk az eszközünkkel egy a fordítási idő csökkentését célzó projektben a Nokia Research Centerrel. Másik példa az OpenOffice++ EU finanszírozású projekt, ahol célunk az OpenOffice.org elemzésével a program architektúrájának feltárása és a forráskód minőségének növelése volt.

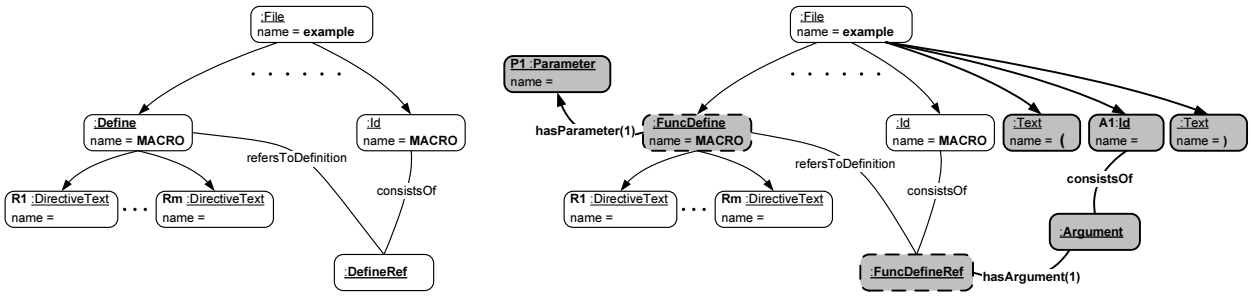
Saját hozzájárulás

A séma és a hozzá tartozó API a szerző munkáját képezi. A modellépítés megvalósítása a szerző munkája, de a forráskód elemző technológia és a séma példányok építési stratégiái a Columbus C++ elemzőjén alapulnak, így ezek közös eredménynek számítanak. A tézispont eredményeit a [19, 20] cikkekben publikáltuk.

I/2 Makrók modell szintű újraszervezése

Annak ellenére, hogy a C/C++ programok újraszervezése (refactoring) gyakori téma a szakirodalomban, preprocesszor direktívák újraszervezéséről alig található publikáció. A modell szintű újraszervezés előnye, hogy a modellek különböző feltételek formális ellenőrzésen is átesnek, ami különösen fontos amikor egy magas szintű újraszervezési művelet több konkrét formája és létezhet. Itt első hozzájárulásunk megfelelő szempontok összegyűjtése makrókkal kapcsolatos magasabb szintű újraszervezési minták konkrét kidolgozásához. Alternatívaként meg kell gondolni C/C++ programelemek (konstansok, függvények), illetve a változó paraméterszámú (variadic) makrók

használatát. A preprocesszálás során fontos szerepe van a fordítási környezetnek is, például a parancssori definícióknak vagy a standard makróknak, mely tényezők általában az előfeltételek közé kerülnek. Más nyelvekhez hasonlóan itt is szükséges a hívó helyek bejárása és módosítása a konzisztencia megőrzése végett. A makró újraszervezés konkrét formáját befolyásolja a makró típusa: tudni kell, hogy objektum- vagy függvény típusú, esetleg változó paraméterlistás makróról van-e szó. A megadott szempontok alapján részletesen is kidolgoztuk és tárgyaltuk a makrókhoz paraméter hozzáadását célzó újraszervezési műveletet. Gráf transzformációs megoldást választva [15, 14] megadtuk a transzformáció bal és jobb oldalát. Objektum típusú makróhoz paraméter hozzáadását végző újraszervezésre egy példa látható a 3. ábrán.



3. ábra. Paraméter hozzáadása objektum típusú makróhoz – a transzformáció bal és jobb oldala

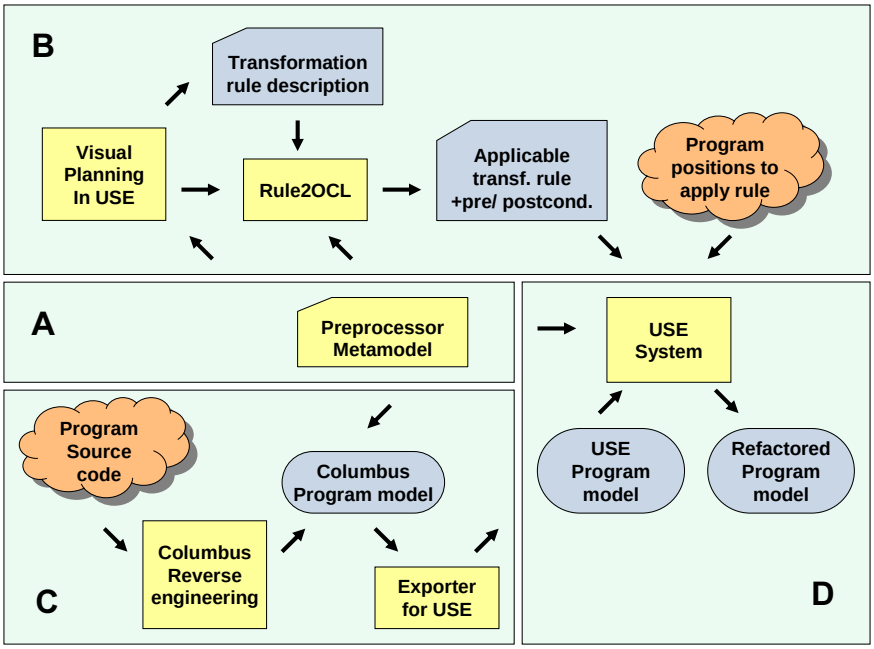
Munkánk érdekessége hogy visszatervezésből származó programokon dolgozunk. Eszköz architektúrát terveztünk, melynek implementációja – főleg meglévő eszközökre támaszkodva – alkalmas makrók újraszervezésének tervezésére (hasznos a magas szintű újraszervezési sémák konkretizálásánál), végrehajtására és annak ellenőrzésére. Az általunk javasolt eszköz architektúra a 4. ábrán látható.

A preprocesszor metamodell (A) szerepe minden főbb lépésnél fontos. A transzformációs szabályt a bal- és jobboldali modellek megadásával tervezzük meg a USE rendszerben (B). A modellek alapján a szabály-leíró fájl félig kézi munkával készül el (későbbiekben automatizálható lépés). A Rule2OCL eszköz a metamodell és a szabály-leíró alapján végrehajtható szabályt generál OCL nyelven, melyhez elő- és utófeltételek is automatikusan készülnek. A kezdeti program modellt a Columbus keretrendszer preprocesszora állítja elő (C). Megoldottuk a séma példányok automatikus konvertálását, hogy azokat a USE UML specifikációs eszköz fel tudja használni, amely eszköz a transzformációkat kezeli [4, 8]. A USE rendszer ellenőrzi hogy a modell megfelel-e a metamodellnek (D). Ezután a paraméterül kapott pontokon végrehajtja a transzformációkat, miután a kapott modellt ellenőrzi és az esetleges hibákat jelzi.

A módszert gyakorlati kísérletekkel validáltuk. Objektum-típusú makrókhoz adtunk paramétert két lépésben: először a definíciót, majd a hívási helyeket is átírtuk. A kísérleteket nyílt forráskódú programokon végeztük el, minden arra alkalmas ponton automatikusan végrehajtva a módosításokat. Tapasztalataink szerint az első implementáció közepes méretű programok esetén alkalmazható, a transzformációk mellett hasznos a modelleknek és magának a metamodellnek az ellenőrzésére is.

Saját hozzájárulás

A fenti eredmények a szerző munkáját képezik és a [18] cikkben kerültek publikálásra. A munka előzménye egy C++ környezetben közreadott publikáció [23]. Az előzményben felhasznált C++ metamodell nem a szerző munkája, a metamodell alapon végzett modell szintű transzformációk alapötlete közös eredmény, míg a konkrét C++ újraszervezési műveletek kidolgozása a szerző munkája.



4. ábra. Refactoring eszköz architektúra

II Szeletelési módszerek módosítások hatásanalíziséhez

A második részben a makrók elemzési eredményeit a programszeletelés területén alkalmazzuk. A módosítások kezelése a szoftver karbantartás sarkalatos pontja. A változás-hatásanalízis során feltárjuk egy adott változtatás továbbgyűrűző hatásait. Egy jól ismert módszer a hatásanalízis támogatására a program szeletelés [25, 26]. Általánosan fogalmazva a szeletelés egy program egy részét (szeletét) határozza meg, mely bizonyos számításokban részt vesz. A szeletelés területe igen szerteágazó, sokféle szeletelési módszer és stratégia létezik. Ezek közös tulajdonsága azonban, hogy a makrókat nem tekintik program pontnak, ami a szeletelés alapvető eleme. A függőség alapú szeletelési módszerek alkalmazásakor egy ún. Program Függőségi Gráf vagy Rendszer Függőségi Gráf épül (PDG vagy SDG – Program/System Dependence Graph) a szeletek kiszámításához [10].

Az általunk bevezetett függőség alapú makró szeletelést és alkalmazását két lépésben ismertetjük. A tradicionális függőség alapú szeletelés alapötletét felhasználva megalkottuk a Makró Függőségi Gráfot (továbbiakban MDG – Macro Dependence Graph). A függőségi gráfon definiáltuk az előrehaladó és hátrahaladó szeleteket. Ezzel lehetővé tettük, hogy olyan kérdésekre is választ adjunk amit a C/C++ szeletelés nem képes: meg tudjuk határozni hogy egy makró definícióban történő változtatás a program mely részeire van kihatással. Második lépésben integráltuk a két fajta (makró és C/C++) függőségi gráfot, definiáltuk az összekötési pontokat hogy az eddigi C/C++ szeleteket makró szeletekkel folytathassuk. Definiáltuk magukat az összekapcsolt előrehaladó és hátrahaladó szeleteket is, melyek globális kiszámítására algoritmusokat adtunk meg. Az elméleti eredmények alátámasztása céljából kidolgoztunk egy eszköz architektúrát, és annak megvalósításával valós programokon értékeltük ki az eredményeket.

Egy egyszerű példával világítjuk meg motivációnkat. Keressük meg egy C/C++ program azon pontjait, amelyeket érint egy makró definíció megváltoztatása. A megváltozott makrót több másik makró törzséből is hívhatják, míg végül, több makróhelyettesítés után része lesz egy (vagy több) C/C++ nyelvű kódrészletnek. Ezek a kódrészletek kihatással lehetnek a program más részeire is, ennek felderítésében már használhatjuk a hagyományos C/C++ szeletelést. Tehát ha a kezdeti makró definícióból indított előrehaladó makró szeletet folytatjuk a megfelelő helyen egy előrehaladó C/C++ szelettel, akkor megkapjuk a teljes keresett programrészletet.

```
1 #define ASSIGN(v) = v
2 #define SGN unsigned
3 #define DECLI(name, val) SGN int name ASSIGN(val);
4 DECLI(i,2)              // ⟹ unsigned int i = 2;
5 printf("%u\n",i);
```

2. forráskód lista. Példa szeletek összekapcsolásához

Az alapötletet vázlatosan a fenti kódrészleten keresztül ismertetjük. A szeletelési kritérium az 1. sorban található definíció, az innen indított előrehaladó makró szelet az 1., 3. és 4. sort tartalmazza. A 4. sorban látható makró hívás fogja összekötni a kétféle szeletet. A preprocesszálás után a `DECLI(i,2)` hívás az `unsigned int i = 2;` kódrészletet eredményezi, ami már C/C++ nyelvi elem. Ez lesz a kritériuma a C/C++ szeletelésnek, amelyhez tartozó szelet a 4. és 5. sorokból áll. Az összekapcsolt szelet tartalmazza az 1., 3., 4. és 5. sort, ami azt jelenti hogy az 1. sor módosítása ezekre lehet kihatással. Ha csak a C/C++ programelemekkel foglalkoznánk, akkor ezek felderítése sikertelen lenne.

A szeletek összekapcsolása a másik irányban is lehetséges. A 3. forráskód listában az előző programkód preprocesszált változatát láthatjuk. Legyen a szeletelési kritérium az `i` változó az 5. sorban. A hátrahaladó C/C++ szelet csak a 4. és 5. sorokat tartalmazza.

```
1
2
3
4 unsigned int i = 2;
5 printf("%u\n",i);
```

3. forráskód lista. Példa forráskód preprocesszálás után

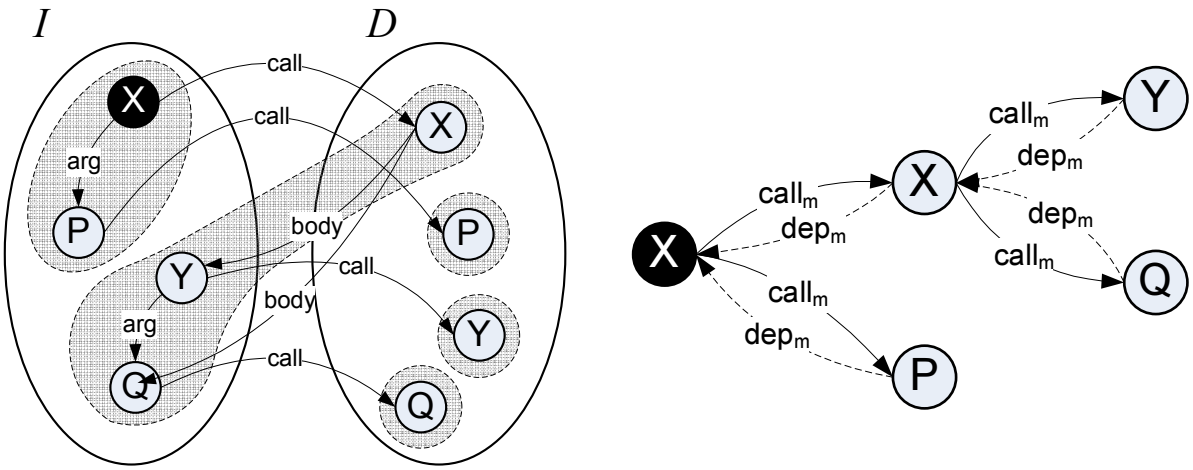
Kihasználva a tényt hogy a 4. sor makró helyettesítésből keletkezett, hátrahaladó makró szelet számításával pontosíthatjuk az eredményt. Az összekapcsolt szelet már tartalmazza mind a 5 sort az eredeti programkódban. Ez az irány felhasználható például abban az esetben, ha hibakeresés közben egy makróhívásra akadunk, s látni szeretnénk pontosan honnan jönnek a hibát okozó C/C++ kódrészletek.

A következőkben három tézis részletesen is bemutatja eredményeinket a szeletelés területén. Ide tartozik a makró szeletelés bevezetése, a C/C++ és makró szeletek összekapcsolása, és az ezekhez kapcsolódó eredmények gyakorlati kiértékelése.

II/1 Makró szeletelés

Amint láttuk, a makróval kapcsolatos információ két irányban is felhasználható. Egy adott makróhívás kapcsán természetes módon felmerül a kérdés hogy pontosan mely definíciók vesznek részt a teljes makrókifejtésben. Ez a leginkább felhasznált irány, a hívás felől a definíciók felé, a korábban említett makró felfedés/elrejtési mechanizmus is ide tartozik. Azonban karbantartási környezetben a másik irány talán még fontosabb: a program mely részeire van kihatással egy makródefiníció megváltoztatása?

Az intuitív módszer végigkeresni a forráskód könyvtárát egy `grep` jellegű eszközzel. Ez több okból sem célravezető, de a fő indok a megközelítésből fakadó hibás eredmény, ami több tényezőnek is köszönhető: (1) include direktívák és konfigurációk, (2) makrók újradefiniálása, és (3)



5. ábra. (a) Makróhívások elemeinek gráf reprezentációja , and (b) A $call_m$ és a dep_m relációk az egyszerűsített makró struktúrán

rejtett makróhívások a ## operátorral. Munkánk során kidolgoztunk egy módszert mellyel választ kaphatunk a fenti típusú kérdésekre.

A makróhívások jól reprezentálhatók gráfokkal. Az 5. ábra bal oldalán egy példát láthatunk makró makróhívások gráf reprezentációjára a makrók törzsével és a makrók közti kapcsolatokkal. Az ábra jobb oldalán ugyanaz a konstrukció látható egyszerűsített formában, a makróhívási reláció($call_m$) mellett a függőségi reláció (dep_m) is fontos szerepet játszik a makrók szeletelésében.

A hagyományos C/C++ szeletelés ötleteinek átültetése a hasonlóságok és különbségek alapos tanulmányozása után lehetséges. A megfelelő alaphalmaz és relációk definíciói után a Makró Függőségi Gráf (MDG) megalkotásának egy fő akadálya maradt. Egy makróhívás eredménye nem a definíció helyétől függ, hanem a hívás helyétől. Ezért minden azonosító egy makró definíciójában makróhívássá válhat a program egy későbbi pontján. Ha az említett azonosítóra vonatkozó makró definíció megelőzi az eredeti makró hívását, akkor az eredeti makró törzse egy újabb hívást is tartalmazni fog. Ezért tehát egy makró definíciója többféle eredményt is adhat a hívás helyétől (kontextustól) függően. A gráfban a függőségi élek színezése biztosítja a szeleteléshez szükséges tulajdonságokat, így teljes szoftver projektek függőségi gráfja is felépíthető szeletelési célokra, nem csak egyes fordítási egységeké.

A függőségi gráfon definiáltuk az előrehaladó és hátrahaladó makró szeleteket. Az előrehaladó és hátrahaladó irányok fogalmát szintén interpretálni kellett. C/C++ nyelv esetén a függőség például a függvények hívásával megegyező irányú, míg makrók esetén ez fordított. Amíg a hívott függvény függ a hívótól, addig makrók esetén a hívó függ a hívott makró definíciótól. Az elnevezések első látásra zavarónak tűnhetnek, de így alkalmasak a szeletek összekapcsolására is, amiről a következő fejezetben esik szó. Jelen tézis összegzéseként megjegyezzük, hogy az általunk bevezetett makró szeletelés módszer segítségével komplex makrókkal kapcsolatos kérdések is megválaszolhatók nagyméretű programokban történt változtatás hatásainak vizsgálata során.

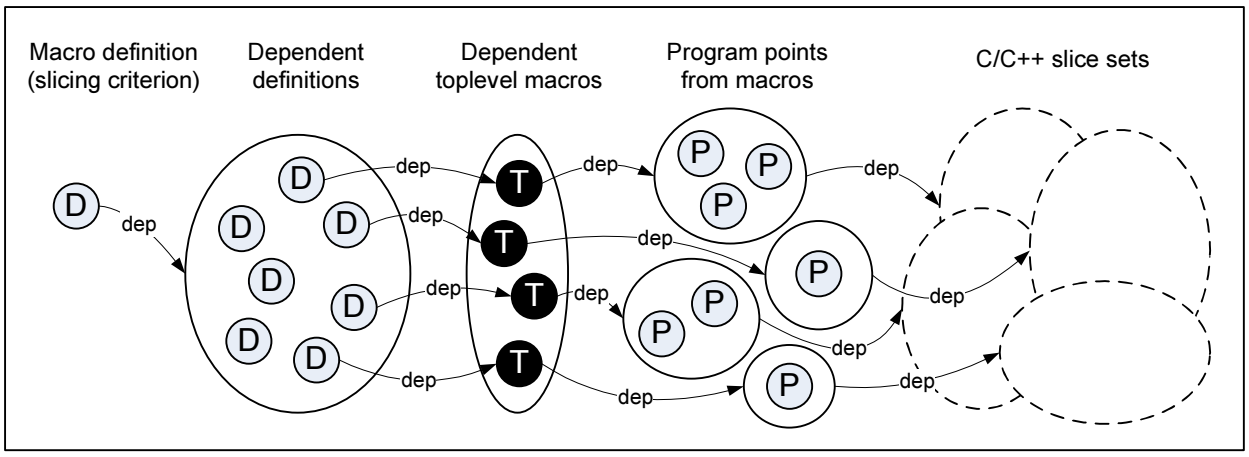
Saját hozzájárulás

A makrók alkalmas gráf reprezentációjának és a csomópontokon definiált relációknak kidolgozása, valamint a Makró Függőségi Gráf megalkotása a szerző munkája. A C/C++ és makró szeletelés fogalmainak diszkussziója, valamint az előrehaladó és hátrahaladó makró szeletek definíciója közös eredmény. A tézispont eredményeit a [21] cikkben publikáltuk.

II/2 C/C++ nyelvű és preprocesszor szeletek összekapcsolása

A makró szeletelés hatásköre önmagában csak a makrókra terjed ki, de a módszer igazi lehetőségeit akkor tudjuk kiaknázni, ha mind a makró szeleteket mind a hagyományos C/C++ szeleteket felhasználjuk. Következő eredményünk hogy a hagyományos C/C++ szeleteket összekapcsoltuk (kombináltuk) a makró szeletekkel, ezzel méginkább teljessé téve a szeletelés alapját képező függőségi halmazt.

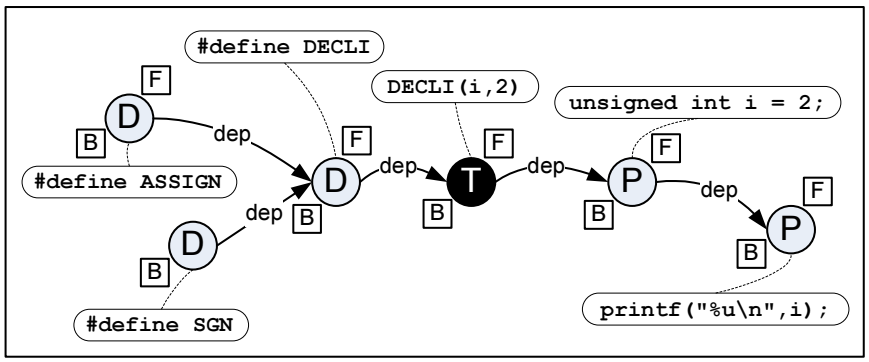
Az összekacsolás alapötletét az előző fejezetben már vázoltuk. A függőség alapú szeletelő algoritmusok a C/C++ szeletelés és a makró szeletelés esetén két diszjunkt alaphalmazon működnek. Az SDG program pontokból áll, mely fogalomnak sok különböző definíciója lehet, számunka fontos tulajdonsága hogy a C/C++ program pontok egymást átfedhetik a forráskódban. Például a 3. forráskód lista 4. sorában egy változó deklaráció és egy értékadás is található. Az SDG program pontjai mindemellett nem tartalmaznak makrókat. Az összekötés szempontjából fontos, hogy be tudjuk azonosítani azokat a program pontokat amelyek makróhívásokból alkalmával keletkeztek. Ezek a program pontok az SDG részei, de az őket eredményező makrók megtalálhatók az MDG-ben. Mivel egy makróhívás több program ponthoz is adhat kódrészleteket, minden ilyen SDG-beli programpontot összakapcsoltunk a hozzá tartozó MDG-beli makróhívással.



6. ábra. Szeletek előrehaladó összekapcsolása függőségi élekkel

Megalkottuk az összekapcsolt függőségi gráfot, mely tartalmazza mindkét eddig gráfot (SDG és MDG), és a köztük kapcsolatot teremtő függőségi éleket a meglévő függőségek kiterjesztésével. Ez a kiterjesztés a makróhívások és a kifejtett makrókból keletkező program pontok forráskódbeli pozíciója alapján történik. Ehhez a makróhívások részletes elemzése volt szükséges, amit a preprocesszor séma biztosított. Az összekapcsolásban részt vevő elemeket a 6. ábrán figyelhetjük meg. Előrehaladó esetben a kiindulási pont egy makró definíció, a makró szelet különböző definíciókon (D) keresztülhalad az MDG-n, míg végül egy vagy több C/C++ kódbeli makróhívással (T) lezárul. A makróhívásokhoz függőségi élek mentén társítjuk a belőlük legalább részben keletkezett program pontokat (P), és a szeletelés folytatható a hagyományos C/C++ szeleteléssel. Az összekapcsolt függőségi gráfon történő szeletekés közben tehát váltás történik a C/C++ program pontokról preprocesszor elemekre és vissza. A korábban bemutatott példa (2. forráskód lista) alapján készített összekapcsolt gráfot a 7. ábrán láthatjuk. Az 1. sorból indított előrehaladó szelet elemeit `F`, míg az 5. sorból indított hátrahaladó szelet elemeit `B` betű jelöli.

Megadtuk továbbá az összekapcsolt előrehaladó és hátrahaladó szeletek formális definícióját, emellett a számításukhoz szükséges algoritmusokat is. A megadott algoritmusok szeletek globális



7. ábra. Csomópontok és szeletek a példa kódrészlet alapján

számítására alkalmasak, működésüket kis mértékben hozzáigazítottuk a következő fejezetben tárgyalt eszköz architektúrához. Mindezek jelentős előrelépést jelentenek a hagyományos C/C++ szeletelésben, ami még kézzelfoghatóbb az előrehaladó esetben, mert ott a C/C++ szeletelés már magát a szeletelési kritériumot sem érintené.

Saját hozzájárulás

Az összekapcsolt függőségi gráf és a bemutatott szeletelő algoritmusok a szerző munkáját képezik. Az összekapcsolt előrehaladó és hátrahaladó szeletek definiálása közös eredmény. A tézispont eredményeit a [24, 22] cikkekben publikáltuk. A [24] cikk a 2008-as IEEE International Conference on Program Comprehension elnevezésű konferencián, mely a témában világviszonylatban is vezető konferencia, elnyerte a legjobb cikknek járó díjat.

II/3 Szeletelési módszerek gyakorlati kiértékelése

A makró és összekapcsolt szeleteléssel kapcsolatos elméleti eredményeken túl kísérleteket hajtottunk végre a javasolt módszerek gyakorlati kiértékeléséhez.

Makró szeletelés

A makró szeletelő program a séma példányokat felhasználva működik, mivel a séma példányok tartalmaznak minden szükséges információt a szeleteléshez, ellátják a Makró Függőségi Gráf feladatát. A séma példányok ipari méretű programokból is előállíthatóak, első kérdésünk az volt, hogy vajon a makró szeletelés is elvégezhető-e nagy méretekben. A kísérletek ennél talán még fontosabb aspektusa, hogy habár a makrók használatáról elérhető egy igen alapos tanulmány [6], mégsem volt pontos képünk a makró szeletek méretéről, valamint a kis- és nagyméretű szeletek megoszlásáról. A kísérleti futtatásokat a Mozilla Firefox forráskódján végeztük el – ami kemény dió lett volna egy C/C++ szeletelő programnak.

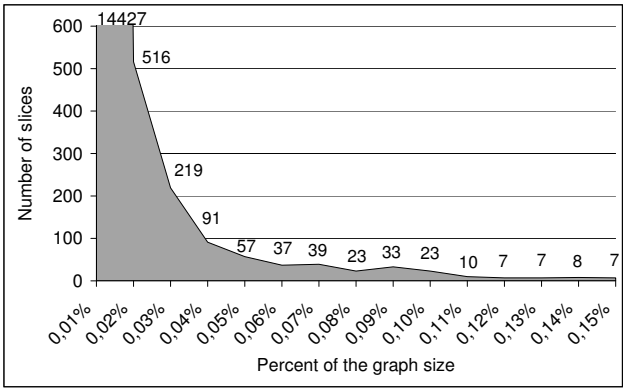
Makró definíciók száma	Hívott makrók száma	Teljes makróhelyettesítések száma
33214	15648	305117

1. táblázat. Makró definíciók és hívások száma

A forráskódban talált makró definíciók és a teljes helyettesítések száma az 1. táblázatban látható. A makró hívások száma igen magas, 90 makró definíciót is több mint 1000 alkalommal hívnak. A 2. táblázatban a definíciónként számolt különálló makróhívások és a makró szeletek méreteinek összesített adatai láthatóak.

	Különálló hívások	Szelet méret
Átlag	53	43
Medián	2	4
Max	47,046	20,040
Min	1	1
Összesen	834,866	674,440

2. táblázat. Makró hívások és szelet méretek definíciónkénti összesítése



8. ábra. Szelet méretek hisztogramja (a gráfmérethez képest %-ban megadva)

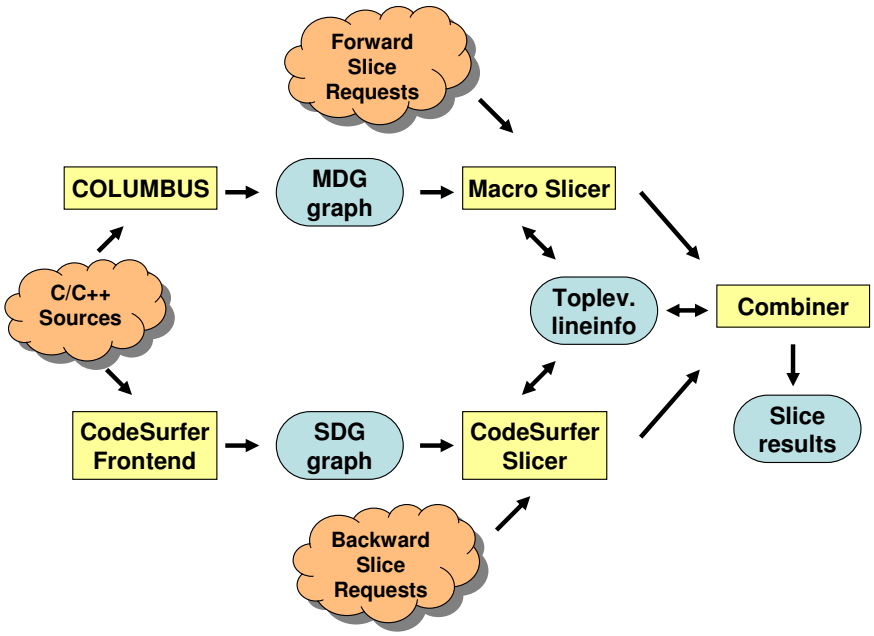
A szeletek méretét a gráf méretéhez hasonlítottuk. Gráf méretként a csomópontok számát tekintettük, ami gyakorlatilag a definíciók és a hívások számának az összege. A relatív szelet méretek hisztogramja a 8. ábrán látható. A hisztogram alakja megfelel az előzetes várakozásunknak. A szeletek döntő többsége kisebb mint a gráf méretének 0,01%-a. Az ábrán az ide tartozó oszlop felülről le van vágva. Emellett 144 darab szelet mérete nagyobb mint 0,15%, ezeket szintén kihagytuk az ábráról, méretük 0,15% és 6,25% között mozog. A szelet viszonylag kicsik, mely a módszer használhatóságát mutatja. Emellett látható hogy sok esetben, ahol nagyobb a szelet mérete, a szeletek kézzel való megkeresése reménytelen vállalkozás lenne.

Az ## operátor használata makróhívások létrehozására az egyik olyan probléma, mely kutatásunkat és az eszköz fejlesztését motiválta. 24 olyan makró definíció található, ahol a konkatenáló operátorral hoztak létre makró hívást. Rajtuk keresztül 337 hívás is történt, ami azt igazolja, hogy ez a furcsa konstrukció valóban használatban van valós programok esetén is.

Összekapcsolt szeletek

Eszköz architektúrát terveztünk egy preprocesszor támogatással rendelkező szeletelő program megvalósításához, mely a szeletek összekapcsolásán alapszik. Az implementációnk egyrészt meglévő eszközökön alapszik, egy jól ismert C/C++ szeletelő program (CodeSurfer) és a makró szeletelő programunk vesz részt benne, kiegészítve egy szelet-összekapcsoló eszközzel.

A hagyományos C/C++ szeletek és a makró szeletek kiértékelése során a szeletelési időt és memóriafogyasztást, az átlag- és szélsőértékeket, valamint a szeletméretek arányait figyeltük meg. A kísérleti futtatásokat 28 nyílt forráskódú programon végeztük el, kezdve kis programoktól



9. ábra. Logikai eszköz architektúra – előrehaladó és hátrahaldó szeletelés

közepes méretűig nagyjából 20k programsorral. A kísérletekben résztvevő programok nagy részét alapműnek számító gyakorlati tanulmányok alapján választottuk a program szeletelés [2], és a preprocesszálás területéről [6]. Úgy gondoljuk, hogy az összesen kb. 240k nem-üres programsor elegendő a bemutatott módszerek használhatóságának igazolására. A mérések során felhasznált programok listája és azok alapvető statisztikai adatai a 3. táblázatban láthatók.

Azt tapasztaltuk, hogy a makró szeletek lényegesen kisebbek, mint a C/C++ szeletek ugyanazon a programon mérve (előrehaladó esetben még jelentősebb a különbség). Habár a makró szeletek kisebbek, mégis valódi előrelépést jelent használatuk, mivel dinamikus jellegük miatt pontos adatokat szolgáltatnak. Ebben a témában a dolgozatban bemutatotthoz hasonló gyakorlati tanulmányt nem ismerünk az irodalomban.

Saját hozzájárulás

A makró szeletek kiértékelésével kapcsolatos eredmények a szerző munkáját képezik (makró szeletelés séma példányokon, makró szeletelő tervezése és implementációja, gyakorlati kiértékelés). Az összekapcsolt szeletek kiértékelésével kapcsolatos eredmények a szerző munkáját képezik (eszköz architektúra, implementáció, kiértékelés), kivéve a CodeSurfer plugin implementálását, amely közös munka volt. A tézispont főbb eredményeit a [22] cikkben publikáltuk.

Összegzés

Kitűzött célunk a preprocesszor direktívák jelenlétével nehezített szoftver karbantartási lépések támogatása. Felismerve, hogy több éves fejlesztés és üzemeltetés után bármely program egyetlen teljes és releváns dokumentációja maga a forrásakód, munkánk során forráskód-alapú visszatervezési megközelítést alkalmaztunk. Teljeskörű megoldást adtunk preprocesszorral kapcsolatos információk kinyeréséhez (visszatervezéséhez), mely információkat a program megértés, az újraszervezés és a hatásanalízis támogatására használtunk fel.

Első hozzájárulásunk a preprocessor séma (metamodel), mely kulcsfontosságú a visszatervezési folyamatban. A séma a C/C++ forrásfájlok minden preprocesszorhoz kapcsolódó elemét modellezi, emellett információt szolgáltat a preprocesszor műveleteiről is, mint például a makró

Program neve	Méret (sorok)	MDG építés ideje (mp)	MDG méret (csúcsok)	SDG építés ideje (mp)	SDG méret (csúcsok)	Makró szeletelés ideje (mp)	C/C++ szeletelés ideje (mp)
replace	512	0.28	136	1.18	3205	0.26	7.85
copia	1085	0.45	7	6.13	94390	0.12	208.65
time	1119	1.88	162	4.15	5633	0.26	3.73
which	1246	1.87	146	5.41	7449	0.48	29.44
compress	1335	0.84	108	2.18	4408	0.16	8.29
wdiff	1364	2.12	217	4.57	7640	0.53	10.77
ed	2637	3.80	117	9.98	39412	0.73	716.82
barcode	2807	6.34	381	13.76	27970	3.1	427.62
tile	3549	1.93	1881	27.69	51095	19.72	146.43
acct	4008	9.37	899	12.50	24619	5.0	116.98
li	4793	10.71	1826	3006.31	943340	79.9	56238.38
EPWIC	5249	12.10	852	14.68	27099	12.23	443.48
lightning	5563	20.8	1750	69.42	56778	6954.21	572.75
gzip	5997	9.88	1725	17.88	37525	34.16	1315.92
userv	6016	5.47	1244	24.72	105902	23.30	3281.28
indent	7582	4.55	857	12.22	42102	17.98	1100.14
bc	9472	9.6	1554	24.90	59503	31.17	2080.13
diffutils	10124	18.91	1971	29.35	53928	31.54	1261.76
gnuchess	11045	13.87	2511	29.12	70782	143.8	4391.19
ctags	11670	12.96	1480	55.31	209357	106.61	12611.60
sed	13339	9.37	2527	26.28	89788	204.76	9374.67
nano	13698	14.96	3964	38.11	177879	591.88	23445.10
ijpeg	15253	25.82	4283	39.75	77531	212.62	6948.48
flex	17533	22.56	3188	112.12	126757	259.55	9912.45
bison	20673	35.74	4387	88.64	138972	98.92	16099.25
wget	21104	27.88	4146	95.28	269209	993.85	60294.88
espresso	21780	3.86	0	52.79	151802	0.18	9642.20
go	22118	5.40	5296	22.18	110236	499.19	22550.61
Összesen	242671	293.32	47615	3846,61	3014311	10326,21	243240,85

3. táblázat. A kísérletekben részt vevő programok

kifejtésről. Tudomásunk szerint ez az első nyilvános, általános célú séma a preprocesszorhoz. Séma példányon (modell) egy gráfot értünk, mely egy adott C/C++ programhoz tartozik és tartalmaz minden preprocesszorral kapcsolatos konkrét információt. A séma konkrét példányaiból mind az eredeti forráskód, mind a preprocesszált forráskód, valamint a preprocesszálási folyamat köztes állapotai is kinyerhetőek. Implementáltunk egy preprocesszor eszközt, amely képes több millió programsoros, ipari méretű programok elemzésére, módszerünket több kutatás-fejlesztési projekt során alkalmaztuk. A séma és a hozzá tartozó programozói interfész (API) lehetőséget biztosít a kinyert információk további célokra történő felhasználására. Ezt a lehetőséget további téziseink során is kihasználtuk.

Legfrissebb munkánkkal makrók újraszervezésére (refactoring) javasoltunk megoldást. A modell szintű újraszervezés előnye, hogy a modellek különböző feltételek formális ellenőrzésen is átesnek, ami különösen fontos, amikor egy magas szintű újraszervezési művelet több konkrét formája és létezhet. Szempontokat adtunk meg makrókkal kapcsolatos magas szintű műveletek végrehajthatóvá tételéhez, mely szempontok alapján konkrét újraszervezési műveleteket dolgoztunk ki. Megadtunk egy lehetséges eszköz architektúrát, melyet meglévő eszközök integrációjával valósítottunk meg. A módszert közepes méretű, valós programokon végrehajtott transzformációval validáltuk.

A dolgozat második részében a szoftver karbantartás egy központi kérdésére kerestünk vá-

laszt: Egy változtatás a program mely részeire lehet kihatással? A program szeletelés fejlesztési és karbantartási feladatokhoz is segítséget nyújt, jól alkalmazható hatásanalízis céljából. Nagy vonalakban a program szelet a program egy pontjához tartozik (ez a szeletelési kritérium), és a program azon részhalmazát jelenti, amely kihatással lehetett a kritériumra. A szeletelés területe igen szerteágazó, sok módszer és szeletelési stratégia létezik. Ezek közös tulajdonsága azonban, hogy a preprocesszor makrókat általában nem tekintik program pontoknak, ami a szeletelés alapegysége. A függőség alapú szeletelési módszerek alkalmazásakor egy ún. Program Függőségi Gráf vagy Rendszer Függőségi Gráf épül (továbbiakban PDG vagy SDG – Program/System Dependence Graph) a szeletek kiszámításához.

Az általunk bevezetett függőség alapú makró szeletelést és alkalmazását két lépésben ismertettük. A tradicionális függőség alapú szeletelés ötleteit kölcsönözve megalkottuk a Makró Függőségi Gráfot, és definiáltuk a rajta számítható makró szeletek mind előrehaladó, mind hátrahaladó formáját. Ezáltal meg tudjuk határozni, hogy egy makró definícióban történő változtatás a program mely részeire lehet kihatással. Második lépésben integráltuk a két fajta (makró és C/C++) függőségi gráfot, definiáltuk az összekötési pontokat úgy, hogy az eddigi C/C++ szeleteket makró szeletekkel folytathassuk. Megadtuk az összekapcsolt előrehaladó és hátrahaladó szeletek definícióját, illetve azok globális kiszámítására algoritmusokat adtunk meg. Az elméleti eredmények alátámasztása céljából kidolgoztunk egy eszköz architektúrát, a megvalósításhoz felhasználtuk a séma példányokat, melyek MDG-ként funkcionáltak. A gyakorlati kiértékelés során valós programok szeleteit számítottuk ki – számítás makró szeletelés esetén hatékonyabb – és elemeztük számítási idejük, méretük és arányaik alapján.

A fent említett témák bőven adnak lehetőséget további kutatások végzésére, melyek közül itt csak néhányat említünk. A séma statikus példányainak generálásával kutatási lehetőség nyílna a konfigurációk területén. Az újraszervezési megoldásunk a változtatások forráskódba történő propagálásával bővíthető. A program megértést támogatandó hasznos lenne a preprocesszor direktívák alkalmas vizualizációja, ezen a területen már tettünk is lépéseket. A makró szeletelés teljes integrálása, részletesebbé tétele szintén jövőbeli kihívást jelent.

A tézisek és a kapcsolódó publikációk (melyek mind első szerzősek) viszonya a 4. táblázatban látható. Összegzésül: a preprocesszált nyelvi környezetben történő program modellezés, újraszervezés és szeletelés terén elért eredményeink – köztük néhány újító megoldás – mind elméleti, mind gyakorlati szempontból elősegítik nagyméretű rendszerek karbantartását.

	Tézis	Publikációk
I/1	Preprocesszor metamodell	[19] [20]
I/2	Model szintű újraszervezés (refactoring)	[23] [18]
II/1	Makró szeletelés	[21]
II/2	Szeletek összekapcsolása	[24] [22]
II/3	Szeletelési módszerek kiértékelése	[22]

4. táblázat. Tézisek és kapcsolódó publikációk

Köszönetnyilvánítás

Hiszem, hogy a dolgozat megírását elsősorban Isten kegyelmének köszönhetem. Emellett sokaknak szeretnék köszönetet mondani, akik valamilyen módon hozzájárultak a dolgozat elkészítéséhez. Először témavezetőmnek, Gyimóthy Tibornak szeretném megköszönni az érdekes kutatási témákat és célokat, a biztos hátteret és a vezetését éveken keresztül. Hálával tartozom Beszédes Árpádnak és Ferenc Rudolfnak, akikkel minden kérdést megvitattunk, a hasznos tanácsaikért és a rengeteg időért amit a közös munkáinkkal töltöttek. Köszönetem fejezem ki kollégáimnak és barátaimnak, néhányukat név szerint is megemlítenék: Szokody Fedornak, Jász Juditnak, Curley David-nek, Siket Péternek, Siket Istvánnak, Frittman Patríciának és Dévai Richárdnak. Szeretném megköszönni Martin Gogolla szíves vendéglátását és támogatását a kutatócsoportjában eltöltött idő alatt. Hálás vagyok barátaimnak Kolozsi Andrásnak, Havasi Ferencnek, Árgyelán Miklósnak és Tasnády-Szeőcs Zoltánnak, és legfőképpen kedvesemnek Horváth Katalinnak türelméért és támogatásáért. Végezetül szeretném kifejezni hálámat családomnak, akik hosszú éveken keresztül mindenben támogattak és bíztattak.

Vidács László, 2009. október

Hivatkozások

[1] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 281, Washington, DC, USA, 2001. IEEE Computer Society.

[2] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of ICSM 2003*, pages 44–53. IEEE Computer Society, September 2003.

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

[4] Fabian Büttner and Martin Gogolla. Realizing graph transformations by pre- and postconditions and command sequences. In *ICGT*, pages 398–413, 2006.

[5] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software 7*, pages 13–17, January 1990.

[6] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12), Dec 2002.

[7] Alejandra Garrido. Program refactoring in the presence of preprocessor directives. Ph.D. thesis, University of Illinois at Urbana-Champaign, USA, october 2005.

[8] Martin Gogolla, Fabian Büttner, and Duc-Hanh Dang. From graph transformation to OCL using USE. In *AGTIVE*, pages 585–586, 2007.

[9] Ric Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, November 2000.

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

[11] Bernt Kullbach and Volker Riediger. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 3–12. IEEE Computer Society, 2001.

[12] P.E. Livadas and D.T. Small. Understanding code containing preprocessor constructs. In *Proceedings of IWPC 1994, Third IEEE Workshop on Program Comprehension*, pages 89–97, Nov 1994.

[13] C. A. Mennie and C. L. A. Clarke. Giving meaning to macros. In *Proceedings of IWPC 2004*, pages 79–88. IEEE Computer Society, 2004.

[14] Tom Mens. On the use of graph transformations for model refactoring. In *GTTSE*, pages 219–257, 2006.

[15] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

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

[17] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *USENIX Summer Technical Conference*, pages 185–197, June 1992.

[18] László Vidács. Refactoring of C/C++ Preprocessor constructs at the model level. In *Proceedings of ICSOFT 2009, 4th International Conference on Software and Data Technologies*, pages 232–237, July 2009.

[19] László Vidács and Árpád Beszédes. Opening up the C/C++ preprocessor black box. In *Proceedings of SPLST 2003, 8th Symposium on Programming Languages and Software Tools*, pages 45–57, June 2003.

[20] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proceedings of CSMR 2004 (8th European Conference on Software Maintenance and Reengineering)*, pages 75–84. IEEE Computer Society, March 2004.

[21] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Macro impact analysis using macro slicing. In *Proceedings of ICSOFT 2007, Second International Conference on Software and Data Technologies*, pages 230–235, July 2007.

[22] László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with C/C++ language slicing. *Science of Computer Programming*, 74(7):399–413, May 2009.

[23] László Vidács, Martin Gogolla, and Rudolf Ferenc. From C++ Refactorings to Graph Transformations. *Electronic Communications of the EASST (ICGT 2006 Workshop Software Evolution through Transformations)*, 3:127–141, September 2006.

[24] László Vidács, Judit Jász, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with C/C++ language slicing. In *Proceedings of ICPC 2008, 16th IEEE International Conference on Program Comprehension*, pages 163–171. IEEE Computer Society, June 2008. Best paper award.

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

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