Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

ANSI C programmodulok közötti hívások felderítése teszt fedettségi analízishez

BME Méréstechnika és Információs Rendszerek Tanszék

Szolgáltatásbiztos rendszertervezés szakirány

2010/2011. I. félév

Önálló laboratórium 2 beszámoló

Készítette: Fejes Endre (G2MMW3)

Konzulens: Piroska László
Csoportvezető
Robert Bosch Kft.

Belső konzulens: Majzik István
Egyetemi docens
Budapesti Műszaki Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Tartalomjegyzék

1 A probléma

1.1 Motiváció

Az Önálló laboratórium 2 keretein belül a korábban Önálló labor 1 illetve szakmai gyakorlat alatt elkezdett feladatot folytattam.

Biztonságkritikus rendszereknél kiemelt fontosságúak és a fejlesztési időigényét nagyban befolyásolják a regressziós tesztek. Ezek időigényessége onnan fakad, hogy mindent hibajavításnál alapértelmezés szerint szükséges lenne az egész rendszer újratesztelése. Ez elkerülhető annak meghatározásával, hogy egy adott helyen történő módosítás az alkalmazás mely további részeit érinti, így kizárhatóak azok a tesztek, amik nem érintettek a változás által. A nem érintett tesztek kijelöléséhez fontos információ, hogy adott tesztek a kód mely részét fedik le.

A célkitűzés egy olyan eszköz tervezése és fejlesztése volt, ami az egyes tesztek kódlefedettségének meghatározásához nyújt segítséget. A feladat konkretizálása során a vizsgálandó kódfedettséget leszűkítettük a programmodulok közötti hívások fedettségére. Ennek megfelelően a támogató eszköz elsődleges célja a modulok közötti hívások felderítése volt.

A Bosch Kft. ennek az eszköznek a használatával az automata sebességváltók mikrokontrollerén futó beágyazott, ANSI C nyelvű programok teszteléséhez szükséges idő lerövidítését tűzte ki célul.

1.2 A konkrét feladat

A lefedettség meghatározásának támogatását az eszköz a program modulok interfészének illetve a modulok közötti, interfészeken keresztül történő hivatkozásoknak a felderítésével végzi.

Az előző meghatározásban rögtön olyan fogalmak kerülnek elő, ami *ANSI C* nyelvben nincsenek definiálva, ezek a *modul* illetve annak *interfésze*. Az adott szoftver esetén a modul a forrásfájlok egy olyan halmazát jelenti, amik egy funkciót valósítanak meg, azonban ezek se nevükben, se tartalmukban, se fizikai elhelyezkedésükben nem utalnak arra, hogy melyik modulhoz tartoznak, ez csupán elvi szinten meghatározott. A másik ilyen fogalom a modul interfésze. Mivel ez sincs nyelvi szinten definiálva, így ezt is szükséges meghatározni. Interfész alatt azokat a fejléc fájlokban leírt függvényeket illetve globális változókat értjük, amikre más modulokból hivatkozás történik. Utóbbi szűkítése az interfész jelentésének azért szükséges, mert a fejlécfájlok olyan elemeket is tartalmazhatnak, amik csak modulon belül megosztottak, és ezek irrelevánsak esetünkben.

A feladat tehát egy C nyelvű, moduláris szerkezetű alkalmazás moduljai esetén az interfészek meghatározása. A készítendő eszközzel szemben támasztott követelmény volt a könnyű használhatóság.

2 Irodalomkutatás

A feladat meghatározása után irodalomkutatást végeztem. Ennek célja elsődlegesen a jelenleg elérhető, a feladatot esetlegesen lefedő kész termékek keresése volt. Azonban mivel ilyet nem sikerült találnom, tovább folytattam a kutatást, a célkitűzésnek megfelelő eszköz fejlesztéséhez szükséges technológiák után.

2.1 Elérhető eszközök

Először kutatómunkát végeztem, hogy felderítsem, nincsenek-e erre a feladatra már kész, felhasználható eszközök. Főleg a *QAC* illetve *Lint* eszközöket vizsgáltam meg. Ezek rendkívül változatos és kiterjedt ellenőrzési mintákat szolgáltatnak, de mindkettő kereskedelmi termék, ráadásul utóbbi körülményesen bővíthető és önmagában nem tartalmaz a szükséges információkat visszaadó funkciókat.

Mivel nem találtam az elvárásoknak megfelelő kész eszközt, ezért a kutatást a saját eszköz megalkotását lehetővé tevő technológiákkal folytattam.

2.2 Szoftver ellenőrzési megközelítések

A szoftver ellenőrzési megközelítések korábbi tanulmányaim okán már nem voltak ismeretlenek számomra, ezért erről csak a dolgozat teljessége miatt következik egy rövid összefoglaló.

Az ellenőrzéseket megkülönböztetjük az alapján, hogy dinamikusak vagy statikusak. A kettő közti alapvető különbséget az jelenti, hogy míg az első a vizsgálatot a program futása közben végzi, az utóbbi a program kódját, annak forrását és egyéb komponenseit elemzi.

A két megközelítés nem zárja ki egymást. Ellenkezőleg, lehetőség szerint érdemes mindkét fajta ellenőrzést alkalmazni. Ennek az a legfőbb oka a tökéletességre törekvésen kívül, hogy egyik megközelítés se tud teljes bizonyossággal működni. A statikus ellenőrzés a kód vizsgálatát jelenti. Az ilyenfajta ellenőrzés a forráskód szisztematikus átvizsgálását igényli, és megmutatja a szintaktikailag lehetséges eseteket. Azonban akár egy feltételes elágazás esetén annak eldöntése, hogy a program melyik ágon futhat tovább, statikus ellenőrzéssel nehézkes. Az ilyen jellegű vizsgálatokra alkalmasabb a dinamikus ellenőrzés, azonban annak is megvan a hiányossága. Ez pedig az, hogy egyszerre csak egy adott futás alatt bejárt állapot sorozatot vizsgál. Azaz anélkül, hogy a programot minden paraméterének minden lehetséges kombinációjával kipróbálnánk, nem állíthatjuk teljes bizonyossággal adott esetek kizárását.

Ez jelen esetben azt fogja jelenteni, hogy a statikus ellenőrzés során megjelenhet olyan elem interfészként, amire semmilyen esetben sem történik *tényleges* hivatkozás, mert a hivatkozó részre

nem kerül a vezérlés. Továbbá azt, hogy kimaradhatnak olyan elemek, amikre olyan – tipikusan dinamikus – módon történik hivatkozás, ami a pusztán a kódot vizsgálva nem meghatározható.

Elméletben persze lehetséges mindkét módon teljes ellenőrzést végezni, azonban ez statikus ellenőrzés esetén egy, a fordító (vagy script nyelvek esetén az értelmező) képességeit is magába foglaló rendkívül bonyolult alkalmazás lenne. Dinamikus tesztelésnél pedig már kevés paraméter esetén is kezelhetetlenül sok tesztesetet jelentene.

Az, hogy adott esetben a vizsgálandó alkalmazás egy beágyazott rendszer szoftvere, további nehézségeket okoz. Statikus ellenőrzésnél egy alacsonyabb szintű, hardver közelibb nyelvet kell feldolgozni, ami, például egy konkrét memóriacímre való hivatkozást véve, sokkal nehezebben követhető, mint egy magasabb szintű nyelv, mint például a C# vagy a Java.

A dinamikus ellenőrzést is bonyolítja a beágyazott környezetben való munka. A futtatás közbeni vizsgálatot legtöbbször a kód valamiféle naplózással kiegészített, módosított változatán szokás végezni, ami lehetőséget ad a rendszer egyszerű megfigyelésére. Azonban egy olyan esetben, amikor futás közben esetleg semmilyen megjelenítő eszköz, és tipikusan csak pár bit vagy bájt memória terület áll rendelkezésre, egy olyan naplózásnak a megtervezése, ami releváns adatokkal szolgál, önmagában is komoly kihívás.

A statikus ellenőrzés a kódot vizsgálja, vagy közvetlenül, vagy annak valamilyen absztrakt reprezentációját elemezve. Ez utóbbi eset egy változata az *Abstract Syntax Tree* (AST) technológia. Ez nem más, mint egy nyelv meta-modelljét leíró szerkezet, amelynek elemeinek használatával egy nyelv elemzése lehetséges. Ebből egy konkrét programot leíró példány a *Concrete Syntax Tree* (CST), amely gyakorlatilag egy objektum orientált reprezentációja a kódnak

2.3 *Eclipse* technológiák

A feladat leírásánál említettem, hogy elvárás volt a könnyű kezelhetőség. Ennek a kitételnek az eszköz egy fejlesztőkörnyezetbe integrált megvalósításával kívántam eleget tenni.

Mivel adott volt egy korábban általam fejlesztett hasonló eszköz architektúrája és terve, ami hasonló feladatra készült, és egy *Eclipse* beépülő volt, ezért ezt választottam platformnak. További érv volt, hogy az *Eclipse* több olyan technológiát is ad, ami az adott eszköz megvalósítását illetve az elemzési feladatokat egyszerűsíti.

2.3.1 *SDK, JDT, PDE*

Az *Eclipse* alapú fejlesztéshez készült *Eclipse* disztribúció az *Eclipse SDK*. Ennek alapja az *Eclipse Platform*, ami a rendszer alapját képezi, és biztosítja a felhasználói felületet. A Platformhoz kapcsolódik a *Java Development Tools* (JDT), ami az *Eclipse* Java fejlesztő funkcióit nyújtja. Az *Eclipse*

ugyanis létezik egyéb programnyelvekhez igazított változatban, például *C++*, vagy *PHP* programozáshoz. Mivel az *Eclipse* maga is Java alapú, ezért a kiegészítők fejlesztését támogató *Plug-in Development Environment* (PDE) a *Platformon* kívül függ a JDT komponenstől is.

A PDE lehetőséget ad az *Eclipse* elérhető funkcióinak kiegészítésére.

2.3.2 *AST, CST és CDT*

A program statikus analízisét *Abstract Syntax Tree* (AST) technológiával oldottam meg. A fenti bevezetésnek megfelelően az AST egy olyan fa, ami egy adott programnyelven írt forráskód absztrakt szintaktikai struktúráját írja le. Minden eleme a forrás egy alkotóelemét reprezentálja. Azért nevezik absztraktnak, mert nem teljes részletességgel adja vissza a forráskódot – bár az én általam használt eszköz erre lehetőséget nyújt.

Az *Eclipse* nyújtotta *C Deveplopment Tools* (CDT) eszközkészletben található AST könyvtár nem más, mint a C illetve C++ nyelv meta-modelljét leíró szerkezet illetve az elemzéshez szükséges osztályok gyűjteménye. Ennek használatával egy C nyelvű kódot tudunk modellezni. Találhatóak benne többek közt a változó és függvény elemeket reprezentáló osztályok illetve a közöttük fellépő kapcsolatokat, mint a függvényhívást leíró osztályok.

A CDT a CST bejárására *Visitor* minta alapú lehetőséget biztosít, ami leegyszerűsíti a különféle nyelvi elemek (pl. függvény deklaráció, értékadás, stb.) különböző kezelését. Ezzel lehetséges a fejléc fájlok-béli deklarációk és kód fájlokban lévő hívások illetve hivatkozások kigyűjtése.

3 Tervezői nehézségek és döntések

A következő alfejezetekben az irodalomkutatás alapján a konkrét feladattal kapcsolatos tervezői döntésekről, illetve még tervezési szinten megjelenő problémákról lesz szó.

3.1 Modulok meghatározása

Mint azt már korábban tárgyaltam, az egész feladat a szoftver moduljainak vizsgálatát célozza meg, annak ellenére, hogy ilyen fogalom az adott nyelvben nem is létezik. A megoldáshoz mégis szükségünk van arra, hogy egy-egy állományról eldöntsük, melyik modulhoz tartozik. Mivel ennek egyértelmű jelölése nyelvi szinten nem megoldott és az állományok elnevezéséből illetve elhelyezkedéséből csak sok és sokféle potenciális kivétel figyelembe vételével lehetne meghatározni, ezért az állományok modulhoz tartozásához az ellenőrzési folyamat más lépései miatt rendelkezésre álló *modulonkénti állománylista* felhasználását választottam.

Ezt azért is tehettem meg, mert a cégnél egy hasonló lista modulonként már rendelkezésre állt, főleg a fordítási folyamat miatt. Ebből már megoldható volt a szükséges információk kinyerése.

3.2 Makrók kezelése

A C nyelvben és különösen beágyazott rendszereknél rendkívül népszerű a makrók használata. Ez érthető, mivel többek között egyszerűsíti a szoftver karbantartását, a külső hivatkozások kezelését, és sok esetben csökkenti a kód méretét. Ezek egymásba ágyazása megengedett, valamint paraméterezhetőségük miatt elég bonyolult a szintaktikájuk. Mivel rengeteg hivatkozás történik ezeken keresztül, ezért szükséges volt ezek feloldására valamilyen megoldást találni.

Ehhez a CDT nem ad segítséget. Saját makró feldolgozó készítése nehéz, de főleg időigényes feladat lett volna, ráadásul nem vetekedhet a fordítók sok éve tesztelt, beépített eljárásával, hiszen gyakorlatilag a fordító előfordítási funkcióját kellene megvalósítani. Ez pedig nem csak rendkívül összetett, de fölösleges munka lenne, mivel ez a funkció már tökéletesen implementált magában a fordítóban.

Szerencsére, mint a legtöbb fordító esetében, a cégnél használt fordítónál is elérhető az előfordítás önálló futtatása. Ez azt jelenti, hogy a normál fordítási eljárást mindössze a fordítási parancsok egy kapcsolóval való kiegészítésével megkaphatjuk a forrásfájlok olyan változatát, amelyben a makrók már be vannak helyettesítve.

Ennek előnye természetesen, hogy egy megbízható, kész makró feldolgozást kaptam. Azonban két negatívuma is van ennek a módszernek. Először is, mivel az `include` parancs is az előfordítónak szánt utasítások közé tartozik, ezért az egész előfordított forráskód egy fájlba kerül. Mivel a feladat a különálló fájlok illetve fájl halmazok vizsgálata, ezért ez nehézséget okoz. Azonban a fordító jelöléseket hagy az összefésült kódban arról, hogy az adott program részlet melyik fájlból származik. Ez alapján visszakövethető az egyes program elemek helye.

A másik negatívum, hogy a makrók önmagunkban is tartalmaznak logikát. Ez azt jelenti, hogy az előfordítás során a dinamikus ellenőrzésnél már tárgyalt problémával kell szembesülnünk: az előfordítás kimenetele függ bizonyos paraméterektől, így csak akkor kaphatunk teljes képet, ha a paraméterek minden érvényes kombinációját lefedjük. Ezt azért tehetjük meg itt, mert egyrészt kevés az előfordítást befolyásoló paraméter van, másrészt ezek egy adott szoftverkonfiguráció esetén elég konstansnak tekinthetőek.

Az előfordítással kapott forráskódot használva tehát az általam fejlesztett eszköz bemeneteként már nem szükséges a makrók kezelése, elégséges a változókra illetve függvényekre koncentrálni.

3.3 Mutatók elemzése

A C nyelv szintén nehézkesen elemezhető nyelvi elemei a mutatók. Ezeknek a feloldása azonban nem szükséges, mivel a mi esetünkben az a lényeg, hogy egy adott interfész elemre hol történik hivatkozás.

Egy modul mutatón keresztül is hivatkozhat egy másik modulban található elemre. Ebben az esetben a mutató beállítását már hivatkozásnak tekintjük. Erre azért van szüksége, mert a mutatóval később végezhetők olyan műveletek, amik követése nehézkes, viszont lehetővé teszik azt, hogy a mutató később esetleg más elemet címezzen meg. Mivel ezek a műveleteket nem túl gyakran, elsősorban tömbök esetén használják, ezért ennek vizsgálata nem feltétlen térülne meg. Ez azért tehető meg, mert gyakorlatilag csak a kizárólag memória lapú címzések felderítése marad ki. Ha például egy függvény mutatókból álló tömb elejét jelző mutató módosításáról van szó, akkor csak a korábban a tömbbe eltárolt címzésű függvények érintettek, amik az eltárolásukkor felderítésre kerülnek.

4 Megvalósítás

4.1 Integráció a fejlesztői környezetbe

Az Önálló Labor 1 idejének nagy része alatt, így a tervezési szakasz elején még nem volt se hozzáférésem, se rálátásom a tényleges tesztelési folyamatra illetve az ahhoz használt eszközökre, amibe később az általam fejlesztett eszközt kell integrálnom. Sajnos, amikor sikerült megismerkednem a rendszerrel, kiderült, hogy a teszteléshez használt belső fejlesztésű *Esprit* keretrendszer batch jellegű folyamatok futtatásán alapszik, így az a koncepció, hogy a forrásállományokat egy *Eclipse* projektként tekintjük, igencsak körülményesen megoldható. Azonban mivel az eszközt egyelőre kevesen használnák csak, így annak átalakítását különálló program formájára későbbre halasztottuk.

Mivel az Önálló Laborok időkerete letelt és a kevés felhasználó miatt elfogadható volt a fenti eljárás, végül ez a megoldás végleges is lett.

Így tehát az integráció úgy valósul meg, hogy a keretrendszerben a fordítási folyamat közvetlenül a fordítás előtt áll meg, a különféle konfigurációs állományok feldolgozása és fordításhoz szükséges segéd fájlok előkészítése után, és ekkor manuálisan szükséges az *Eclipse* indítása, benne egy projekt létrehozása és azon az eszköz futtatása. Azonban így az előfeldolgozás nem történt még meg, így a fordítási lépés elé be kell szúrni a keretrendszerbe egy olyan lépést, ami pontosan ugyanazt végzi el, mint az eredeti fordítási művelet, de a fordítót a csupán előfordításra utasító kapcsolóval hívja meg.

4.2 Az előfordított fájlok feldolgozása

Mint azt már korábban említettem, az ellenőrzés a forrásfájlok fordító által előállított, előfordított változatán történik. Az előfordítás során több forrásfájl egy fájlba történő összefésülése következhet be. Meghatározott jelöléseket használ az előfordító a tagolásra, ezeket az eszköznek kell tudnia értelmeznie és „visszafordítania” fájlokra és hozzájuk tartozó program elemekre.

Ezek a jelölések tulajdonképpen nem mások, mint az aktuális kódrészlet származási helye fájl szerint megjelölve. Egy egyszerű, 2 fájlból álló példa programból generált előfordított fájl tartalma:

```
# 13 "a.h" 2

externint A_GlobalVar_i;
externint A_GlobalVar_a[10];

void A_G2L();
void A_L2G();
void A_FOO(char a);
# 2 "a.c" 2


int a_LocalVar_i;
int A_GlobalVar_i;
int A_GlobalVar_a[10];
void (*a_LocalVar_p)(char);

int main(void) {
int main_LocalVar_i;
 a_LocalVar_p = A_FOO;
 a_LocalVar_p(65);

 a_LocalVar_i = 1;

 A_FOO(30 +41);
return 0;
}

void A_FOO(char a) {
 printf("%c\n", a);
}

void A_G2L() {
 a_LocalVar_i = A_GlobalVar_i;
}

void A_L2G() {
 A_GlobalVar_i = a_LocalVar_i;
}
```

Megfigyelhető, hogy gyakorlatilag az eredeti fájlneveket tartalmazó megjegyzések a fájlokhoz tartozó kódrészleteknek megfelelő intervallumokra bontják így az összefésült fájlt. Mivel ennek elemzése során szükség lesz egy adott sorban megjelenő deklaráció illetve hivatkozás eredeti fájlhoz

társítására, ezért az elemzés első lépéseként egy intervallum fát épít az eszköz, ami a sorszámokkal indexelve a fájlneveket tartalmazza.

A fájl soronkénti olvasása közben a határoló sorokat az alábbi reguláris kifejezés segítségével azonosítom:

```
static String MARKER_PATTERN_STR = "^#\\s*[0-9]+\\s*\"([^\"]+)\".*";
```

4.3 Változó deklarációk, függvény definíciók felismerése

A statikus ellenőrzés feladata a modulok közötti interfészként használt vagy használható változók illetve függvények összegyűjtése. Ez a korábban említett interfész meghatározás alapján azt jelenti, hogy először is a fejlécekben definiált elemek halmazára van szükségünk.

A CDT által nyújtott AST osztályokból épített CST lehetőséget ad *Visitor* minta alapú bejárásra. Ez azt jelenti, hogy a heterogén kollekcióként elérhető C nyelvi elemeket tartalmazó CST bejárható úgy, hogy a függvények túlterhelését kihasználva egy bejáró *(Visitor)* objektum magát a bejárandónak *(Visitable)* átadja, és az egy visszahívást végez a bejárón, így az éppen látogatott *Visitable* konkrét típusának megfelelő túlterhelt függvénye futhat le.

A deklarációk deklaráló részét leíró nyelvi elemek az `IASTDeclarator` interfészt valósítják meg, azon deklarátorok pedig, amik függvényt deklarálnak az `IASTFunctionDeclarator` interfészt. A függvény deklarátoroknál a nevét leíró gyermekelem használatával már regisztrálható a függvény deklaráció, egyéb esetben a deklarációt meghatározó elemek vizsgálatával megállapítható, hogy változó, enumeráció, struktúra vagy egyéb deklarációról van szó, és így meghatározhatóak a változók is.

4.4 Hivatkozások változókra, függvényekre

A definíciók felderítése után a következő lépés a definiált interfész elemekre való hivatkozások megállapítása. Ehhez az előzőkhöz hasonlóan az AST nyújtotta lehetőségeket használtam fel. A CST bejárása során a definiált elemekre való hivatkozást az `IASTName` interfész írja le, így az ilyen szintaktikai alkotórészeket kellett megvizsgálni. A függvényhívások esetében például az `IASTName` szülő eleme minden esetben `IASTFunctionCallExpression`, ez alapján felismerhető és rögzíthető hivatkozásként.

A hivatkozások azonosításánál a C nyelv egyik sajátossága, a gyakori mutatóhasználat jelenti a legnehezebb feladatot. Megengedett az alábbi kódrészletben mutatott módon egy típustalan mutatót bármilyen függvényre állítani és utána azt megfelelő típuskonverzióval meghívni.

```
void *fnPointer_p;
void foo (char a) {
    printf("%c\n", a);
}
fnPointer_p = (void *)foo;
((void (*)(int))fnPointer_p)(65);
```

Az ilyen típusú függvény illetve változó hivatkozások visszakövetése statikus analízissel rendkívül összetett feladat, mivel ilyen esetekben a hivatkozásnál csak a mutatót lehet közvetlenül megállapítani, a hivatkozott elem meghatározásához vissza kell követni a mutató értékadásait. Ez azonban nem okoz gondot, mivel amikor egy változó vagy függvény címét értékül adjuk egy mutatónak, akkor szintén történik hivatkozás az adott elemre és ez rögzíthető.

Problémát jelenthetnek azonban az eszköz felhasználási célterületére, a beágyazott rendszerekre fejlesztett szoftverek sajátosságai. Konfigurációk beolvasása történhet olyan módon, hogy a konfigurációt adó változók a beágyazott rendszer memória területének ismert pozícióján találhatóak és a programban ennek címe számértékkel van átadva egy mutató értékéül. Az ilyen hivatkozásokat az eszköz nem kezeli, de ezek az esetek ritkák és ismertek, és alapvetően külön figyelmet szánnak az ellenőrzésükre.

Itt meg kell jegyezni ismét a statikus és a dinamikus ellenőrzés adta két lehetőség egymást kiegészítő szerepét. A statikus ellenőrzésnél kimaradhatnak olyan hivatkozások, amelyek dinamikus úton jönnek létre. Azonban megtalálhat olyanokat, amelyek esetleg semmilyen körülmények között nem aktiválódnak, mert egy adott (esetleg hibás) programrészlet miatt nem fut rájuk a vezérlés. Ezzel szemben a dinamikus ellenőrzésnél azok a hivatkozások maradhatnak ki, amelyek nem egy olyan állapot sorozat alatt érintettek, amely lefutásának feltételeit a tesztelés paraméterei lehetővé teszik. Ugyanakkor futás közben még a legösszetettebb, legmélyebb hivatkozás is egyszerűen megfigyelhető.

Elmondható tehát, hogy a hivatkozások keresése statikus illetve dinamikus ellenőrzéssel két, egymást metsző halmazt ad meg. A metszeten kívüli elemek száma az ellenőrzések pontatlanságával arányos, azonban ezt a pontatlanságot nem feltétlenül lehet hatékonyan csökkenteni.

Az adott esetben a dinamikus ellenőrzés a cég szoftvereinek tesztelése során valósulhat meg, a statikus ellenőrzés pedig a jelen Önálló Labor feladat témája. Az erre fejlesztett eszköz célja megállapítani az összes olyan lehetséges hivatkozást, amit a tesztelések során érinthet a program. Későbbi lépés lehet a dinamikus ellenőrzés kiegészítése oly módon, hogy elérhető legyen a futás közben ténylegesen érintett elemek listája.

5 Dinamikus ellenőrzés

A statikus ellenőrzés által megállapított interfész elemek listájának korábban tárgyalt módon való megerősítését illetve kiegészítését teszi lehetővé a dinamikus ellenőrzés. Ennek a műveletnek az előkészítése valójában a statikus ellenőrzés alatt történik, mivel a kód módosítása szükséges hozzá.

A módosítás lényege a kód felműszerezése, azaz olyan utasításokkal történő kiegészítése, ami a megfigyelhetőséget biztosítja. Mivel beágyazott rendszereknél változó és igen korlátozott lehetőségeink vannak a naplózásra, ezért egy olyan felműszerező folyamat szükséges, amiben könnyen felüldefiniálható az, hogy pontosan mik is legyenek a beszúrt utasítások. Alapértelmezett működésként – feltételezve, hogy a teszt során valamiféle szöveges konzol rendelkezésre áll, - a beszúrt utasítások a lehető legtöbb információt az alapértelmezett kimenetre naplózzák.

A hivatkozásoknak az alábbi típusait különíthetjük el, a felműszerezés megvalósítását figyelembe véve.

- Függvényhívás
- Függvény vagy változó értékül adása
- Értékadás változónak

Függvényhívás esetén a függvények elejére történő naplózó utasítások beszúrásával már hasznos információkat kaphatunk, azonban ez kimerül a függvény meghívásának tényében illetve egyfajta hívási gráf kirajzolódásában.

```
void A_FOO(char a) {
 printf("%c\n", a);
}
```

```
void A_FOO(char a) {
LOG_FUNCTION_CALL("A_FOO");
 printf("%c\n", a);
}
```

Itt természetesen a `LOG_FUNCTION_CALL` illetve `LOG_PARAMETER` makrókat az adott környezetnek megfelelően lehet definiálni.

Ennél azonban több információt nyerhetünk, ha a másik két esetnél adódó egyetlen reálisan megvalósítható és univerzálisan működő módszert választjuk, ez pedig az utasítást megelőzően való beszúrása a naplózó parancsnak. Megjegyzendő, hogy a jelen feladatnál csak az érintett metódusok illetve változók azonosítása a cél, így a függvények paramétereinek értékének illetve a függvények visszatérési értékének rögzítése szükségtelen (és a beágyazott működést tekintve nem is lenne praktikus). Ebben az esetben több dologra kell ügyelni:

- Az utasítás, amiben a hivatkozás történik, lehet több soros is. Ez azt jelenti, hogy nem feltétlen ténylegesen a fájlban megelőző sorba kell végezni a beszúrást, hiszen ez a működés megzavarásához, vagy hibájához vezet.
- Egy sorban több utasítás is található. Ilyenkor a rendszer működésének folyamatát nehéz vizsgálni, mivel az utasítások kiértékelési sorrendje a jellegüktől, és kapcsolatuktól függően történik. Esetünkben mivel elég az tudnunk, hogy történt-e hivatkozás vagy nem, a sorrend informatív, de nem szükséges információ, ezért csak arra kell ügyelni, hogy az egy sorban szereplő *összes releváns* utasítás naplózva legyen.

Sajnos a *CDT* a vizsgált fájlok módosítására nem ad interfészt. Azonban annyi segítséget nyújt, hogy a *Visitor* bejárás során elérhető *CST* elemeknek megadja a fájlban való elhelyezkedését. Ez azt jelenti, hogy bár a felműszerezést „kézzel” kell végezni, de az az információ elérhető, hogy hol szükségesek a beszúrások.

Érdemes arra figyelni, hogy fájlonként a végétől visszafele végezzük a módosításokat, így a még el nem végzett módosítások helye nem változik.

6 Az eszköz minőségének ellenőrzése

Mivel az egész feladat biztonságkritikus rendszerek ellenőrzésének támogatását szolgálja, ezért szükségszerűen magának az eszköznek is fontos a minősége. Ennek biztosítására az eszközt mind a *PMD* mind a *FindBugs* eszközök minden lényeges ellenőrzésének megfelelő módon készítettem el, illetve javítottam az implementációt.

7 Az eszköz működése

Az eszköz egy *Eclipse pluginként, Jar* formátumban készült el. Ezt az *Eclipse plugin* nevű mappájába másolva telepíthető. Ezután a *C/C++* projektek helyi menüjében az alábbi ábrán látható módon válik elérhetővé a funkciói.

1. ábra
Eszköz integrációja a helyi menübe

A *CStaticAnalyzer* nevű menü alatt a következő opciók érhetőek el.

7.1 Előfordított fájlok analízise *(Analyze Precompiled Source)*

Ez a funkció végzi a korábban leírt módon az analízist: a projektben előfeldolgozott fájlokat keres (ezeket *„.pre”* kiterjesztésük alapján azonosítja), majd ezeket veti alá elemzésnek. Feldolgozza a modul fájl lista *(„.mfl”*) állományokat, amik alapján az egyes fájlok modulokhoz rendelését végzi el.

Az elemzés végeztével egy jelentés generálódik. Ez modulonkénti és fájlonkénti bontásban listázza az azonosított interfész elemeket. Először a modulhoz tartozó fejléc fájlokban definiáltakat, jelölve, hogy melyik másik modulban, és azon belül fájlban történt az adott elemre hivatkozás. Ezután a modulhoz tartozó fejléc és forrás fájlokban talált hivatkozásokat más modul interfész elemeire.

Egy egyszerű mintaprogram modul leíró fájljait, forráskódját, előfordított forrásállományát és az elkészített jelentést a Függelék tartalmazza. Megtalálható még a fordításhoz illetve előfordításhoz használt segédfájl *(Makefile)*. Érdemes megfigyelni ezen, hogy a kétfajta parancs között csak kis különbség van, így a ennek a műveletnek az integrációja a fordítási folyamatba egyszerű.

7.2 Forrásfájlok analízise *(Analyze Source)*

Amennyiben az előfordítás nem megoldható, az eszköz lehetőséget ad az eredeti forrásfájlok alapján elemzést végezni. Azonban ez azt jelenti, hogy a makrók nem kerülnek feloldásra. Ez jelentősen csökkentheti az eredmény relevanciáját, így ez az ellenőrzési mód nem javasolt, ha a makrók gyakoriak a kódban.

7.3 Gyorsító tár ürítése *(Clear Cache)*

Az eszköz minden lefutás elején inicializálja az adatszerkezetet. Előfordulhat azonban olyan eset, hogy ez valamilyen hiba folytán meghiúsul. Ez esetben, ez a művelet ezzel az opcióval, kézzel is elvégezhető egy elemzés előtt, így garantálva, hogy az eszköz az *Eclipse* újraindítása nélkül is megbízható jelentéseket generáljon ismételt ellenőrzések során.

8 Összefoglalás

Az Önálló Labor során sikerült az eszköz megtervezése és olyan szintű implementációja, ami meghatározza, és egy jelentésben összesíti egy előfeldolgozott ANSI C moduláris program interfészeit. Szintén elkészült az eszköz végleges folyamatba való integrálásának terve, aminek megvalósításával már most hasznos információkat lehet kigyűjteni egy adott alkalmazásról.

Az Önálló Labor második féléve során sikerült az eszköz statikus ellenőrző részének véglegesítése illetve a dinamikus ellenőrzés rész megtervezése.

Függelék

A. Modul fájl listák (modul fájl lista formátum)

A.a *test/module_a.mfl*

```
C:\test\a\a.c /SRC
C:\test\a\a.h /DEF
```

A.b *test/module_b.mfl*

```
C:\test\b\b.c /SRC
C:\test\b\b.h /DEF
```

B. Fordítási segédfájl: *test/Makefile*

```
clean:
	del *.exe *.pre

all: a/*.c b/*.c
	gcc -o test.exe a/*.c b/*.c

pre: a/*.c b/*.c
	gcc -E a/*.c b/*.c > precomp.pre
```

B. Forrás fájlok

B.a *test/a/a.h*

```
#ifndef A_H_
#define A_H_

#include "../b/b.h"

extern int GlobalVar_i;

void foo (char a);
void KisMicsoda (int i_u8);
void A_G2L();
void A_L2G();

#endif /* A_H_ */
```

B.b *test/a/a.c*

```
#include "a.h"

int a_localVar_i;
int GlobalVar_i=0;
void *fnPointer_p;


int main (void) {

     int mainLocalVar;

     printf("Hello World\n");
     KisMicsoda(65);
     foo('x');
     printf("GlobalVar_i: %d\n", GlobalVar_i);
     A_G2L();
     printf("GlobalVar_i: %d\n", GlobalVar_i);
     a_localVar_i = 3;
     A_L2G();
     printf("GlobalVar_i: %d\n", GlobalVar_i);
     B_L2G();
     printf("GlobalVar_i: %d\n", GlobalVar_i);
     B_G2L();
     printf("GlobalVar_i: %d\n", GlobalVar_i);
     GlobalVar_i++;
     printf("GlobalVar_i: %d\n", GlobalVar_i);
     fnPointer_p = (void *)foo;
     ((void (*)(int))fnPointer_p)(66);
     return 0;
}

void KisMicsoda (int i_u8){
     foo((char)i_u8);
     return;
}

void foo (char a) {
     printf("%c\n", a);
}

void A_G2L() {
     a_localVar_i = GlobalVar_i;
}

void A_L2G() {
      GlobalVar_i = a_localVar_i;
}
```

B.c *test/b/b.c*

```
#include "b.h"

int b_localVar_i=2;


void B_G2L() {
     b_localVar_i = GlobalVar_i;
}

void B_L2G() {
      GlobalVar_i = b_localVar_i;
}
```

B.d *test/b/b.h*

```
#ifndef B_H_
#define B_H_

extern GlobalVar_i;

void B_G2L();
void B_L2G();

#endif /* B_H_ */
```

C. Előfordított fájl: *test/precomp.pre*

```
# 1 "a/a.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "a/a.c"
# 1 "a/a.h" 1
# 11 "a/a.h"
# 1 "a/../b/b.h" 1
# 11 "a/../b/b.h"
extern GlobalVar_i;

void B_G2L();
void B_L2G();
# 12 "a/a.h" 2

extern int GlobalVar_i;

void foo (char a);
void KisMicsoda (int i_u8);
void A_G2L();
void A_L2G();
# 2 "a/a.c" 2
```

```
int a_localVar_i;
int GlobalVar_i=0;
void *fnPointer_p;


int main (void) {

 int mainLocalVar;

 printf("Hello World\n");
 KisMicsoda(65);
 foo('x');
 printf("GlobalVar_i: %d\n", GlobalVar_i);
 A_G2L();
 printf("GlobalVar_i: %d\n", GlobalVar_i);
 a_localVar_i = 3;
 A_L2G();
 printf("GlobalVar_i: %d\n", GlobalVar_i);
 B_L2G();
 printf("GlobalVar_i: %d\n", GlobalVar_i);
 B_G2L();
 printf("GlobalVar_i: %d\n", GlobalVar_i);
 GlobalVar_i++;
 printf("GlobalVar_i: %d\n", GlobalVar_i);
 fnPointer_p = (void *)foo;
 ((void (*)(int))fnPointer_p)(66);
 return 0;
}

void KisMicsoda (int i_u8){
 foo((char)i_u8);
 return;
}

void foo (char a) {
 printf("%c\n", a);
}

void A_G2L() {
 a_localVar_i = GlobalVar_i;
}

void A_L2G() {
  GlobalVar_i = a_localVar_i;
}
# 1 "b/b.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "b/b.c"
# 1 "b/b.h" 1
# 11 "b/b.h"
extern GlobalVar_i;

void B_G2L();
void B_L2G();
# 2 "b/b.c" 2

int b_localVar_i=2;


void B_G2L() {
```

```
 b_localVar_i = GlobalVar_i;
}

void B_L2G() {
  GlobalVar_i = b_localVar_i;
}
```

D. Jelentés

```
Module: C:\test\module_b.mfl
    Global methods defined in C:\test\b\b.h:
        B_G2L
            referenced in module: C:\test\module_a.mfl
            referenced in file: C:\test\a\a.c
        B_L2G
            referenced in module: C:\test\module_a.mfl
            referenced in file: C:\test\a\a.c
    Referenced variables in C:\test\b\b.c:
        GlobalVar_i
            defined in module: C:\test\module_a.mfl
            defined in file: C:\test\a\a.h
Module: C:\test\module_a.mfl
    Global variables defined in C:\test\a\a.h:
        GlobalVar_i
            referenced in module: C:\test\module_a.mfl
            referenced in file: C:\test\a\a.c
            referenced in module: C:\test\module_b.mfl
            referenced in file: C:\test\b\b.c
    Referenced functions in C:\test\a\a.c:
        B_G2L
            defined in module: C:\test\module_b.mfl
            defined in file: C:\test\b\b.h
        B_L2G
            defined in module: C:\test\module_b.mfl
            defined in file: C:\test\b\b.h
```