



Eötvös Loránd Tudományegyetem, Informatikai Kar,
Programozási Nyelvek és Fordítóprogramok Tanszék

Mintaillesztési módszerek vizsgálata absztrakt szintaxisfákon

TDK Dolgozat
2013

Horváth Gábor
Programtervező Informatikus BSc,
II. évfolyam

Témavezető: **Pataki Norbert**, tanársegéd

Tartalomjegyzék

1. Bevezetés	3
1.1. Statikus analízis	3
1.2. STL analízise	4
2. A mintaillesztés háttere	6
2.1. Irodalmi áttekintés	6
2.2. STL-hez kapcsolódó hibák keresése	7
2.3. Eszköz kiválasztása	8
3. Tipikus hibák a STL használatában	11
3.1. A módszer által ellenőrzött hibák	11
3.2. További hibalehetőségek	14
4. Implementációs részletek	16
4.1. Architektúra	16
4.2. Predikátumok	18
4.2.1. Bool vector	18
4.2.2. COAP	19
4.2.3. Find és count	20
4.2.4. Reverse iterator	21
4.2.5. Funktor paraméterek típusa	21
4.2.6. Állapottal rendelkező allokátorok	22
4.2.7. Állapottal rendelkező predikátumok	23
4.2.8. Polimorfikus funktorok	24
4.2.9. Ürességvizsgálat	24
4.2.10. Shrink to fit	25
4.2.11. Remove-val kapcsolatos hibák	26
4.2.12. Másoló algoritmusok és iterátor adapterek	27
4.2.13. Polimorfikusan használt konténerek	28
4.2.14. Tiltott temporálisok	28

5. Távlatok	30
5.1. Implementációt érintő fejlesztések	30
5.2. Az elmélet továbbfejlesztése	31
6. Összefoglalás	32

1. fejezet

Bevezetés

1.1. Statikus analízis

Egy program szintaktikus helyessége nem ad semmiféle garanciát arra, hogy a kód szemantikailag is helytáll. A szemantikus helyesség bizonyítása nehéz feladat, és általában (bizonyos alkalmazási területektől eltekintve) nem éri meg a belefektetett időt és energiát [2]. A helyességbizonyítás hiányában elkerülhetetlenek a programhibák. A kódok komplexitása megnehezíti a hibakeresést, és ez nagyban befolyásolja a programozók produktivitását. Ezért jelent problémát az a tendencia, hogy az idő múlásával a kódbázisok mérete és bonyolultsága is nő. Többek között ez az oka annak, hogy egyes vállalatoknak megéri olyan eszközöket vásárolni, melyek statikus analízist hajtanak végre a kódjaikon. Az eszköz által talált hibák javításával megspórolható az az idő, amit ezek keresésével kellene tölteni.

Egyik fontos alkalmazási terület a C és C++ kódok elemzése, mivel ezen nyelvek felépítéséből adódóan sok a hibalehetőség. A nyelvek alacsony szintje miatt a hibák egy része is elég alacsony szintű ahhoz, hogy viszonylag egyszerű legyen olyan mintákat definiálni, amik illeszkednek rájuk.

Statikus analízist a modern fordítók is végeznek, figyelmeztetéseket adnak a szemantikailag gyanús kódrészletekre. A leggyakoribb hibák mintáit keresik, amiknek egy része elgépelésből származik. Az első különálló ellenőrző program, ami a *lint* volt, már a 70-es évek végén megjelent. Ez is mutatja, hogy milyen hamar felmerült rá az igény. Részben emiatt hívják ezeket az eszközöket lintelőknak vagy lint eszközöknek. Ma már széles körben bevett gyakorlat, hogy a fordító mellett egyéb eszközöket is felhasználunk diagnosztikai célokra. Jelenleg a legnépszerűbbek: *PC-lint*, *PVS-Studio*, *Cpp-check*. Emellett a nagy cégek, mint például a Google, saját eszközöket fejleszt-

tenek a kódjaik ellenőrzésére. Ezek az eszközök gyakran a cég technológiáira specifikus ellenőrzéseket is tartalmaznak.

A statikus analízis létjogosultságát igazolja például a PVS-Studio által talált hibák listája a Microsoft publikus könyvtáraiban [4]. Ez is jelzi, hogy még azoknál a cégeknél is előfordulhatnak detektálható hibák, amelyek megtehetik, hogy a szakma legjavát foglalkoztatják. Sem a képzettség, sem a gyakorlat nem tud teljesen felvértezni a hibák ellen, amelyeket a fáradtság, figyelmetlenség és a közeli határidők idéznek elő. A Google egyik megoldása erre, hogy minden kódot legalább két fejlesztőnek kell látnia, mielőtt bekerül a tárolókba.

1.2. STL analízise

Kezdetben csak az alacsony szintű hibákat keresték ezek az eszközök, például memóriaszivárgásokat. Ez az oka annak, hogy ezek az ellenőrzések szinte kizárólagosan a nyelvvel kapcsolatos hibákra specializálódtak. Az idő múlásával és a technológiák fejlődésével egyre magasabb szintű hibák megtalálására lettek képesek. A keresett hibák jelentős része Scott Meyers tanácsain alapulnak [6, 8]. Emellett számos elírásból származó hibára is létezik figyelmeztetés. Ugyanakkor a C++ szabványnak nem csupán a nyelv, hanem a standard könyvtárak is a részei [12]. A szabványos könyvtári csomag (STL – Standard Template Library) használatával magasabb absztrakciós szinten történik a programozás, ezáltal csökkenthetjük a klasszikus C/C++ hibák előfordulásának a gyakoriságát, javíthatjuk a kódjaink minőségét és fenntarthatóságát. Széles körben felhasználható és hatékony, ezért nagyon elterjedt. Azonban az STL használatának is megvannak a maga buktatói, melyekről többek között Scott Meyers Effective STL című könyvében is olvashatunk [7]. Emiatt meglepő, hogy az ehhez kapcsolódó tipikus hibák kiszűrésére jelenleg csupán erősen korlátozott eszközök érhetőek el, pedig az elterjedtségéből adódóan igény lenne rá. Emellett a szabványban specifikált viselkedése miatt alkalmas arra, hogy mintákat illesszünk rá.

A probléma adott. A megoldás pedig egy olyan eszköz, amely kifejezetten az STL használatához fűződő hibákat keresi. Statikus ellenőrzés esetén az első lépés a kód parszolása (elemzése). C++ esetében ez nem triviális feladat, hiszen egy-egy token jelentése nagyban függ a környezetétől, így ezt a lépést a Clang fordítóra bízom, ami jó szintaxisfát épít fel a kódból, majd ezen végeztem a statikus analízist. Az elkészült alkalmazást ebben a dolgozatban részletesen be fogom mutatni. Tehát az itt megfogalmazott elmélet mögött implementáció is áll, amit jelenleg a kutatásunkat támogató cégnél, az Ericssonnál használnak fel. Emellett az eszköz fejlesztése folyamán

együttműködtem a felhasznált fordító fejlesztőivel. Az általam implementált kódok egy része a Clang kódbázisába is bekerült ¹. A fordító mintaillesztésre felhasznált technológiáinak a továbbfejlesztésében én is részt veszek.

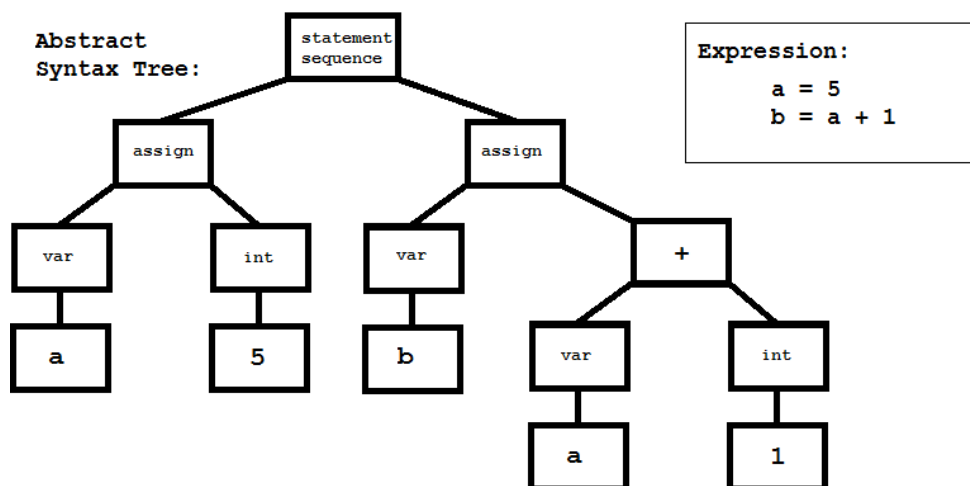
¹Az általam beküldött módosítások: r164943, r164304, r166477, r164123 [17].

2. fejezet

A mintaillesztés háttere

2.1. Irodalmi áttekintés

Az *absztrakt szintaxisfa* (abstract syntax tree, AST) egy adott nyelven írt forráskód ábrázolásmódja (2.1 ábra). A fában minden csúcs (node) megfeleltethető egy konstrukciónak a forráskódban. Ezek a konstrukciók nem feltétlen jelennek meg a kódban explicit módon, erre jó példa az implicit konverzió. Az absztrakt jelző arra utal, hogy a reprezentációban nem található meg a szintaxis minden részlete, például egy kifejezés esetén a műveletek elvégzési sorrendjét a zárójelek is meghatározhatják, amik viszont nem jelennek meg a szintaxisfában node-ként, mivel ez az információ a fa felépítésében van kódolva.



2.1. ábra. Egy példa AST

Számos eszköz képes az absztrakt szintaxisfán különböző műveleteket (pl. refaktorálást, vizualizálást és ellenőrzéseket) végrehajtani.

A CppDepend például egy multifunkcionális eszköz, ami segít C++ kódok vizualizálásában, a kódok megértésében és refaktorálásában [15]. Emellett tartalmaz egy lekérdező nyelvet, amit CQLinq névre kereszteltek. Ez a lekérdező nyelv tulajdonképpen a .Net-es nyelveknél megismert LINQ, ami hasonlít az SQL-re. A segítségével a C++ kódon lehet lekérdezéseket futtatni, például egy osztály publikus metódusait kilistázni.

Voltak kísérletek szintaxis alapú mintaillesztésre is, ilyen volt a SCRUPLE [11]. Definiáltak egy nyelvet magas szintű minták illesztésére. Ez a nyelv azonban nehezen használható és erősen nyelvfüggő volt. Emellett összetettebb nyelvek esetén, mint amilyen például a C++ is, különösen nehéz hasonlót megalkotni.

A Python nyelv standard moduljai között van egy `ast` és egy `parser` modul. Ez megkönnyíti azoknak az eszközöknek az írását, amik Python kódot hivatottak elemezni vagy módosítani. Nem rendelkeznek ugyanakkor ezek a modulok magas szintű műveletekkel, a mintaillesztést visitorok segítségével lehet megvalósítani.

Különböző fejlesztői környezetek SDK-i (Software Development Kit) hozzáférést biztosítanak az AST-hez, azonban a Python modulokhoz hasonlóan nem biztosítanak a mintaillesztéshez magas szintű eszközöket.

2.2. STL-hez kapcsolódó hibák keresése

A statikus analízisnek egy alternatívája lehet, ha egy külön eszköz fejlesztése helyett az STL fordításidejű ellenőrzései kerülnek kiterjesztésre. Ezt a metaprogramozás eszközei teszik lehetővé. A módszer lényege, hogy bizonyos megszorítások adhatóak meg, amelyek ellenőrzése fordítási időben lehetséges [9]. Ezeket a megszorításokat a C++-ban koncepcióknak hívják. A koncepciók implementálása mellett kihívást jelent a felhasználó számára a könnyen értelmezhető hibaüzenetek generálása is.

Ezzel a módszerrel maga az STL könyvtár kerül módosításra. Ahhoz, hogy a felhasználó hozzájusson az analízis eredményéhez, a módosított könyvtárral kell lefordítania az alkalmazását. Nagy kódbázisok esetén bizonyos esetekben ez portolási nehézségekkel is járhat. Emellett az STL implementáció újabb verzióira is portolni kell az ellenőrzést végző kódrészleteket. Ez főleg az új szabvány miatt fellépő módosításoknál okozhat gondokat. Szerencsére a program fejlesztési és kiadási verziója futhat külön könyvtárakon, azonban ez azt is jelenti, hogy kisebb mértékben kerül tesztelésre mind a két könyvtár, ezzel növelve a hibalehetőségek számát.

Az STLint egy másik STL analízissel kapcsolatos eszköz, amit ma már nem fejlesztenek [3]. Megközelítésében nem közvetlen a nyelv elemzésével foglalkozik, hanem az STL-t a nyelv egy kibővítéseként kezeli. Ezt az teszi lehetővé, hogy az STL működése a szabványban a nyelvhez hasonló módon van definiálva. Ez azt jelenti, hogy az analízis alapvetően az STL szemantikáját ellenőrzi a nyelv szemantikája helyett, amiben azt implementálták. Ezáltal az ellenőrzést egy magasabb absztrakciós szinten végzi el.

2.3. Eszköz kiválasztása

Egy kézenfekvő lehetőség statikus analízisre a szöveg közvetlen feldolgozása lenne. Ezt általában reguláris kifejezésekkel szokták elvégezni, ami rendkívül bonyolult, mivel még a fehér karakterek is problémákat okozhatnak. Ez a probléma áthidalható úgy, hogy a szövegszerű mintaillesztési lépést megelőzi egy tokenizálás, és a szövegszerű feldolgozást már a tokeneken hajtjuk végre. Ezt a módszert választotta a *CppCheck* is, azonban ez az eszköz sok hibás eredményt ad. Az AST-n való kereséssel ennél sokkal pontosabb eredmények érhetőek el. Egy C++ parser fejlesztése viszont komplex és időigényes feladat, ezért kézenfekvő egy már működő megoldás felhasználása.

Az LLVM projektre épülő Clang fordítóra esett a választásom [13]. Ez egy nyílt forráskódú, gyorsan fejlődő fordító, amit számos nagy cég támogat, többek között az Apple és a Google. A döntés egyik oka, hogy a fordító felépítéséből adódóan lehetőség van a frontend használatára anélkül, hogy kódgenerálással kapcsolatos függősége legyen az implementációnak [5]. Emellett a Clang API jól dokumentált és kiforrott egy ilyen projekthez. A Clang tervezésében az elejétől fogva fontos szerepet játszott a modularitás, és ezáltal nem csupán egy fordítóra, hanem egy teljes platformmá vált. A döntés meghozatalában nagyban segített, hogy a jelenleg fejlesztés alatt álló verzióban nemrég jelent meg egy új komponens, a LibTooling, ami kifejezetten önálló alkalmazások fejlesztését hivatott megkönnyíteni [14]. A LibTooling könyvtárral egy időben jelent meg egy másik újdonság is, az AST-Matcher könyvtár. Ezt elsősorban refaktorálásra alkalmas eszközök elkészítésére fejlesztették ki, ugyanakkor statikus analízisre is kiválóan alkalmas.

Érdekes megfontolni azokat az érveket, amik a LibTooling fejlesztése alatt szerepet játszottak. Kellett egy módszer, amellyel a mintákat, amelyeket illeszteni próbálunk, definiáljuk. Kézenfekvő lenne, hogy a mintákat definiáló nyelvet a C++ mintájára alkossuk meg, hiszen ekkor mindenki, aki tapasztalt a nyelvben, minimális idő ráfordítása után tudna mintákat létrehozni. Ugyanakkor, ha jobban belegondolunk, ez nem feltétlen jó megközelítés. Nehéz egy ilyen nyelvet úgy megalkotni, hogy a C++ kód minden elemére

mintát tudjunk illeszteni. Tipikusan problémát jelentenek például a destruktork hívások, implicit kasztolások, default konstruktor hívások és a temporális objektumok. Hasonló problémát vet fel az öröklődés is: külön kell választani a közvetlen öröklődést a közvetettől. Tegyük fel, hogy kiindultunk egy C++ közeli nyelvből, hogy egyszerűen megérthessék a C++ programozók. Ha minden szélsőséges esetet figyelembe veszünk, akkor annyi új nyelvi elemet vagyunk kénytelenek bevezetni, hogy csupán az eredeti célunkat nem érjük el, hogy hasonlítson a C++-ra. Emellett egy-egy node tulajdonságaira, és a köztük levő relációkra is akarhatunk mintát illeszteni, például, ha csak olyan kifejezésekre vagyunk kíváncsiak, amikben megtalálható egy részkifejezés, és emellett egy tetszőleges osztály konstruktorában vannak. Ebből következik, hogy egy C++-hoz hasonló nyelv nem alkalmas arra, hogy minták definiálására alkalmazzuk.

Alapvetően a minták állapotmentesek, és jól kombinálhatóknak kell lenniük. Szintén előnyös, ha nagy a kifejezőerejük, ezáltal a minták könnyen olvashatóak. A Google dolgozói ezekből a problémákból és elvárásokból szűrték le azt a konklúziót, hogy funkcionális megközelítést érdemes alkalmazni. Ráadásul azok a függvények, amiket a nyelvben definiálunk, egyértelműen elfogadnak vagy elutasítanak egy node-ot, ezért gyakorlatilag predikátumoknak tekinthetők. Ezeket a függvényeket a továbbiakban *matchereknek* hívom. A funkcionális megközelítés ráadásul lehetővé teszi, hogy a későbbiekben könnyedén párhuzamosítsuk a mintaillesztés folyamatát. Ennek főleg nagyobb kódbázisok esetén lehet jelentősége.

Az ötlet, hogy a fákon való keresést predikátumokkal végezzük, nem újszerű, hasonló megoldásokat láthatunk néhány *unit test* könyvtárban is, például a Google C++ Mocking Framework-ben [16]. Azokat az ötleteket, amelyek ott beváltak, a Clang-es C++ AST *matcherek* tervezésekor is felhasználták.

A *matcherek*et alapvetően három csoportra lehet bontani. Az első csoportot az úgynevezett *node matcherek* alkotják. Ezek képezik a mintáink magját. Minden mintának tartalmaznia kell ebből legalább egyet. Ez határozza meg, hogy mi a típusa annak a *node*-nak, amire kíváncsiak vagyunk. Egy ilyen *matcher* például a `varDecl()`, ami egy tetszőleges változódeklarációt fogad el. Ezt később tovább finomíthatjuk a szűkítő és bejáró *matcherekkel*, amiket argumentumként adunk át a *node matchereknek*. Kényelmi okokból az összes *node matcher* tetszőleges számú paramétert fogad el, és csak akkor fogadja el az adott *node*-ot, ha minden paraméterül kapott *matcher* is elfogadja azt. Miután már ismerjük, hogy milyen típusú *node* az, amit keresünk, a második csoportot alkotó szűkítő *matcherek* segítségével tudunk ennek a tulajdonságaira mintákat illeszteni. A *matcher*-ek típushelyesek, tehát csak akkor fordul le a kód, ha az adott típusú *node*-ról tényleg lekérdezhető a keresett tulajdonság. Emellett elérhetőek általános logikai

szűkítő matcherek is, amik tetszőleges node-okon működnek. Ilyenek például az `allOf`, `anyOf`, `anything` és `unless` matcherek, amikkel egyszerű mintákat tudunk összekomponálni összetettebbekké. Ha az előbb bemutatott matchert akarjuk azokra a deklarációkra leszűkíteni, amik definíciók is, az így nézne ki: `varDecl(isDefinition())`. Az utolsó csoportot az AST bejáró matcherek alkotják. Ezekkel tudjuk a mintán belül a node-ok egymás közötti kapcsolatát definiálni. Tipikusan ilyen kapcsolatot jelent az AST-n belüli elhelyezkedése a node-oknak. Például ha azokra a változódeklarációkra vagyunk kíváncsiak, amik definíciók, és egy függvényhívással vannak inicializálva, az eképp adható meg: `varDecl(isDefinition(), hasInitializer(callExpr()))`.

Egy-egy minta nem feltétlen csak egy-egy node-ra vonatkozik. Talán a legszemléletesebb példa, mikor egy öröklődési láncra akarunk mintát illeszteni. Miután megadtuk a mintát, ami ábrázolja ezt a láncot, azt is meg kell adnunk, hogy a láncnak melyik elemei érdekesek számunkra, amikkel a későbbiekben tovább akarunk dolgozni. Ennek a problémának a megoldására elnevezhetjük a minta egyes részeire illeszkedő node-okat, majd később ezek alapján a nevek alapján tudjuk visszakeresni őket. Az elnevezések ugyanúgy a matcherek részét képezik.

Az így létrehozott minták, ha elfogadnak egy adott node-ot, akkor meghívódik egy *callback* függvény, amelyben a teljes AST elérhető. Amennyiben szükséges, itt lehet elvégezni az AST által reprezentált kód refaktorálását, valamint vizsgálni az AST olyan tulajdonságait, amire a matcherek nem alkalmasak.

A matcherek működésének a lényege, hogy a fordító fordítási egységenként elemzi a fájlokat, és felépíti az absztrakt szintaxisfát. Minden így kapott fára elindít egy rekurzív bejárást, ahol minden egyes node-ot leellenőriz, hogy megfelel-e a mintának. A bejárás alatt több matcher illesztése is megtörténik, így nincs szükség több bejárásra. Jelenleg nincs rá lehetőség, hogy a bejárást csak egy részfára indítsuk el vagy arra, hogy fordítási egységek között végezzünk statikus analízist, ugyanakkor erre az általam fejlesztett eszközhöz nem is volt szükség.

Egy forrásfájl önmagában nem hordoz elég információt a pontos elemzéshez. Ehhez szükségünk van az *include* könyvtárak helyére, a fordítási paraméterekre, és a kívülről definiált makrókra. Ezért a *Tooling* könyvtár számára fel kell építeni egy adatbázist az állományokról és a fordítási parancsokról. Ezt kézzel felépíteni nagyobb projektek esetén fáradtságos és hosszú ideig tartó munka. Ezt orvosolandó az egyetemen futó kutatási projektben kifejlesztettünk egy olyan eszközt, amely tetszőleges make rendszer esetén képes a fordítási paraméterek naplózására. A működésének a lényege, hogy a path környezeti változóba bekerül egy saját `g++` nevű script, ami elvégzi a naplózást, majd meghívja az eredeti fordítót.

3. fejezet

Tipikus hibák a STL használatában

3.1. A módszer által ellenőrzött hibák

A Standard Template Library egy általános felhasználású könyvtár, aminek a jelenlétét és a működését a C++ szabvány garantálja. Alapvetően három nagy csoportra oszthatók a könyvtár által nyújtott komponensek: konténerekre, iterátorokra és algoritmusokra [1]. A konténerek adatszerkezeteket valósítanak meg, az iterátorok ezek elemeinek a felsorolásában, módosításában segítenek. Az algoritmusok konténerfüggetlen műveletek gyakran előforduló feladatokra, mint például rendezés, keresés vagy másolás. Az algoritmusok funktorok segítségével szabhatóak testre, ilyen objektumok segítségével a felhasználói kódrészletek hatékonyan hajthatók végre a könyvtáron belül. Az STL használata számos előnnyel jár, többek között a használatával elkerülhető sok tipikus hiba, továbbá hatékony és platformfüggetlen.

Ebben a fejezetben azokat a tipikus STL jelenségeket fogom bemutatni, amiket az általam fejlesztett eszköz detektálni tud. Azért van szó jelenségekről, mert ezek nem mindegyike jelent egyértelműen hibát, egy részük csupán olyan veszélyes konstrukció, amit az esetek számottevő részében a programozók el szeretnének kerülni. Természetesen előfordulhat olyan felhasználás, ahol ezek indokoltak, épp ezért, az eszközben minden warning kikapcsolható. A továbbiakban minden nem beépített típus vagy függvény, amit megemlítek, `std` névtérbeliek.

Az első ilyen konstrukció a `vector<bool>`, amit a szabvány egy sablon specializációként definiál. A specializáció célja, hogy kompaktabb módon kerüljenek a logikai értékek tárolásra. Ez a szabványban úgy szerepel, hogy

valójában nem a `bool`ok kerülnek tárolásra, hanem egészek, amiknek a bitjei reprezentálják a logikai értékeket. Mivel bitre nem lehet referenciát visszaadni, ezért a `vector []` operátora nem egy `bool` értékkel, hanem egy proxy-osztállyal fog visszatérni. Ez azonban bizonyos körülmények között nem felel meg a szabvány által a vektor felé támasztott követelményeknek, ezért ez a konstrukció kerülendő [7].

Az előzőnél sokkal nagyobb problémát okozhat, ha `container< auto_ptr <T> >` kerül példányosításra, ahol a `container` egy tetszőleges STL konténer. Ezt a konstrukciót gyakran rövidítik a szakirodalomban COAP-nak. Az `auto_ptr` egy *smart pointer*, ami magától felszabadítja az általa mutatott objektum területét, amikor kimegy a hatókörből. Ugyanakkor ezt a pointert úgy tervezték meg, hogy egy objektumra legfeljebb egy `auto_ptr` mutathasson. Ezt néhol úgy fogalmazzák meg, hogy az `auto_ptr` másoló konstruktor destruktív, azaz amikor másolás történik két `auto_ptr` között, a forrás nullpointerré válik. Ez tipikusan akkor jelent problémát, ha bármilyen, potenciálisan másolással járó algoritmust hajtunk végre a konténeren, például rendezést. Ez a konstrukció portolási problémákat is okozhat, ugyanis egyes implementációk esetén az ilyen kód nem fordul le [7].

A `find` egy tetszőleges intervallumban keres meg egy elemet. Az `count` pedig megszámolja, hogy egy adott intervallumban egy elem hányszor szerepel. Ezek az algoritmusok lineáris futási idővel rendelkeznek, nem használják ki az elemek rendezettségét. Ugyanakkor a `set`ben és `multiset`ben az elemek mindig rendezetten vannak tárolva, ezért rajtuk logaritmikus időben is végrehajtható keresés. Ezek a konténerek rendelkeznek `find` és `count` tagfüggvénnyel, ami kihasználja ezt a tulajdonságot, ezért ilyen esetekben érdemes ezeket használni az `algorithm` könyvtár függvényei helyett [7].

A `unique` és `remove` függvények valójában nem törölnek semmit a konténerekből, csupán a végükre másolják a törlendő elemeket, és visszaadnak egy iterátort, ami a legelső törlendő elemre mutat. Ezért ha ezeknél a függvényeknél nem használjuk fel a visszatérési értéket, az az esetek túlnyomó többségében helytelen használatra utal. Másik tipikus hiba a törlő algoritmusokkal kapcsolatban, ha fel is használjuk a visszatérési értéküket a konténer `erase` metódusában, de nem adunk meg egy második iterátort, ami azt jelzi, meddig kell törölni az elemeket. Ekkor nem minden törlendő elemet távolítunk el, csupán az elsőt [7].

Mivel az STL-ben az algoritmusok konténerfüggetlenek, a másoló algoritmusok sem tudnak semmit annak a konténernek a kapacitásáról, ahova másolják az elemeket. Felteszik, hogy a konténerben van elég hely a másolandó elemek számára. Amennyiben ezt nem tudjuk garantálni, segítségünkre lehetnek az iterator adaptorok. Ugyanakkor ezeknek a használatát gyakran mellőzik, és ez futás idejű hibához vezethet [7].

Az STL konténerek esetében lehetőségünk van saját allokatórok megadására, ezzel testreszabható a memóiafoglalás menete. A C++ szabvány szerint az STL implementációk feltehetik, hogy az összes azonos típusú allokatórok objektum ekvivalens és egyenlő. Ebből az következik, hogy egy allokatórnak nem lehet állapota¹. Ennek az az oka, hogy előfordulhat, hogy az egyik konténer allokatóra által létrehozott objektumot egy másik konténer allokatóra fogja felszabadítani. Ezt a jelenséget előídezhetheti a `list splice` művelete.

Az STL számos különböző iterátor típust nyújt számunkra. Ezek közül az egyik a `reverse_iterator`, ami nem konvertálódik automatikusan `iterator`-rá. Azonban rendelkezik egy `base` metódussal, amivel visszakaphatjuk az eredeti iterátort. Ugyanakkor egy érték elcsúszás van az `iterator`ok és a `reverse_iterator`ok között, például előfordulhat, hogy egy `reverse_iterator` egy konténer második elemére mutat, ugyanakkor a belőle kinyert `iterator` a harmadikra. Ezt könnyű figyelmen kívül hagyni, ezért ez a konverzió külön odafigyelést igényel a programozó részéről.

Az `unary_function`ból és a `binary_function`ból leszármazó funktorok esetén a funktor paramétereinek és a visszatérési értékének a típusát több helyen is meg kell adni. Az előbbi osztályok `template` paramétereként és a leszármazott osztály `operator()` függvényének a paramétereinél egyaránt. Ebből újabb hibalehetőségek adódnak, amennyiben ezek a típusok nem konzisztensen lettek megadva.

Az algoritmusok a funktorokat érték szerint veszik át. Emellett nincs megkötés a C++ szabványban arra, hogy egy algoritmuson belül a funktor hányszor másolódhat. Léteznek olyan STL implementációk, ahol problémát okozhat, ha egy predikátumként viselkedő funktor eredménye nem csak a paramétereitől függ [7].

Mivel a funktorok érték szerint adódnak át, ezért nem lehetnek polimorfikusak. Ha egy leszármazott funktort átadunk egy olyan függvénynek, ami a funktor őssztályát várja paraméterül, akkor slicing jelensége jöhet létre [12]. Ennél nagyobb problémát jelent, hogy a virtuális hívások csak referenciákon vagy mutatókon keresztül működnek.

Az STL konténerek sem használhatóak polimorfikus módon, ennek oka elsősorban az, hogy a destruktoraik nem virtuálisak [12]. Ebből adódóan akár memóriaszivárgás is felléphet.

Minden STL konténer rendelkezik egy `empty` függvénnyel, ami megadja, hogy az üres-e. Ez a függvény garantáltan konstans időn belül lefut. Ugyanakkor sokan használják üresség vizsgálatára a `size` tagfüggvényt, ami például

¹A C++11-es szabványban már van rá lehetőség, hogy állapottal rendelkező allokatórokat hozzunk létre.

lista esetében lineáris futási idővel bír². Bár elképzelhető, hogy amikor a programot írták, akkor olyan adatszerkezetet használtak, ahol a `size` futási ideje konstans, de az implementáció az idő folyamán változhat, ezért érdemes arra törekedni, hogy az ürességet mindig az `empty` tagfüggvénnyel vizsgáljuk.

Egy C++11-specifikus probléma, hogy egy `vector` vagy akár `string` méretének a csökkentésére bevett gyakorlat, hogy egy új ideiglenes objektumot hozzunk létre, amibe átmásoljuk az eredeti elemeit, majd kicseréljük a 2 objektumot. Erre a C++11 szabvány egy hatékonyabb megoldást biztosít a `shrink_to_fit` tagfüggvényen keresztül.

3.2. További hibalehetőségek

Ebben az alfejezetben azokat a hibalehetőségeket ismertetem, amiket az eszköz nem ellenőriz, mégis érdemes megemlíteni. Két oka lehet annak, hogy nem kerültek ezek az ellenőrzések implementálásra. Az egyik, hogy nem lehetséges a jelenlegi eszközökkel fordítási időben ellenőrizni az adott problémát, a másik, hogy a felhasznált technológiák limitációja szab gátat.

Gyakori hiba, amibe sok kezdő C++ programozó belefut, az iterátorok invalidálódása. Leggyakoribb eset, amikor egy `vector`-ba új elem kerül, de az kifogyna a tárhelyből, ezért új, nagyobb memóriaterületre mozgatja át az elemeit. A művelet előtt eltárolt iterátorok ezután nem a `vector` által birtokolt memóriaterületre fognak mutatni.

Az STL bizonyos algoritmusainak speciális előfeltételei vannak. Tipikusan ilyen a `lower_bound`, ami csak rendezett konténeren működik megfelelően. A konténer rendezettsége azonban statikus analízis segítségével nem vizsgálható, csak futási időben [10]. Lehetséges lenne ugyan olyan ellenőrzést írni, ami megpróbál leellenőrizni minden lehetséges futást, és jelez ha olyan konténerre hívódhat meg az adott függvény, amin előtte nem volt `sort` hívás. Azonban a `sort` hívásra nincs szükség azokban az esetekben, amikor a konténer elemei egy olyan bemenetről származnak, ahol a rendezettség már garantált. Az ellenőrzés bonyolultsága mellett ezeknél az eseteknél a legtöbb esetben nem lehet kiszűrni az indokolatlan figyelmeztetéseket. Megoldás lehetne egy speciális annotáció bevezetése ami segíti az analízist végző program munkáját, azonban ez irányban a dolgozat írásának idejében nem végeztem fejlesztéseket. Egy másik felmerülő probléma, hogy a `lower_bound` csak akkor ad megfelelő eredményt, ha ugyanazt a rendezést használja, mint a `sort`. Ez viszont nem állapítható meg minden esetben statikus analízissel, mivel függhet futásidejű paramétereiktől is.

²A C++11 szabványban már konstans idejű a listák hosszának a lekérdezése is

Az asszociatív konténerek megkövetelik, hogy a kulcsként használt elemeken definiáljunk egy szigorú részbenrendezést. Amennyiben a definiált rendezés nem felel meg ennek a követelménynek, az nem definiált végeredményhez vezethet. Ezt a tulajdonságot azonban a fordító nem ellenőrzi. A legegyszerűbb eseteket, amikor nem szigorú reláció szerint történik a rendezés, könnyű kiszűrni, ugyanakkor a dolgozat megírása idejében ez az ellenőrzés még nem került implementálásra.

A C++ szabvány nem mond semmit arról, hogy mely fejállományok mely másik állományokat használják, ezért ha könnyen portolható kód a cél, akkor előnyös, ha minden felhasznált STL osztályhoz és függvényhez tartozó fejállomány include-olásra kerül. Ez statikusan könnyen ellenőrizhető, ugyanakkor ez az ellenőrzést nem végezhető el az AST vizsgálatával, mivel az már az előfeldolgozott fájlokból jön létre. Emiatt az általam implementált eszköz nem foglalkozik az ezzel kapcsolatos ellenőrzésekkel.

4. fejezet

Implementációs részletek

4.1. Architektúra

Az első lépés az eszköz használatához a fordítási paraméterek naplózása. Erre egy különálló eszközt fejlesztettünk az egyetemen futó projektben, ami tetszőleges make rendszerrel kompatibilis, és egyszerű a használata. Ennek a pontos működésére ebben a dolgozatban nem térek ki. Az általam fejlesztett program induláskor beolvassa a konfigurációs fájlokat és az előbb említett eszköz naplófájlját. Az összes mintához, amit az AST-n illeszteni akarunk, saját osztály tartozik, amiből egy-egy objektum jön létre induláskor. Ezek az objektumok a rájuk vonatkozó konfigurációkat külön kezelik. Ez a döntés azért született meg, hogy az ellenőrzések bővítésénél minél kevesebb helyen kelljen módosítani a forráskódot. Ezt a célt sikerült megvalósítani, mivel egy új osztály létrehozásán kívül csupán egy belőle legyártott objektum példányosítását kell megtenni, ha bővíteni akarjuk az alkalmazást.

Az ellenőrzésekért felelős osztályok mind egy közös absztrakt osztályból származnak le:

```
struct Predicate {
    Predicate() : _active(false) {}
    virtual ~Predicate() {}

    virtual bool matches(const MatchResult &result_) = 0;
    virtual void configure(std::string conf_) = 0;

    virtual std::string getErrorMsg() = 0;
    virtual std::string getID() = 0;
    virtual std::string getBoundID() = 0;
    virtual MatcherProxy getMatcher() = 0;
    virtual std::vector<MatchPosition>& getExcludes() {
        return _excludes;
    }
    virtual bool isActive() { return _active; }
    virtual void setActive(bool active_) { _active = active_; }
```

```
protected:
    bool _active;
    std::vector<MatchPosition> _excludes;
};
```

Mindegyik osztályhoz tartozik egy egyedi azonosító és egy üzenet, amit a felhasználó felé megjelenítenek, amennyiben találat történik (a predikátum elfogadja az adott AST részletet). Emellett mind rendelkezik egy kivételistaival, ahova a konfigurációk beolvasásakor a hamisnak jelölt találatok kerülnek. Ezek nem lesznek többet jelentve a felhasználó felé. Az `isActive` függvény szerepe azért fontos, mert minden ellenőrzés (továbbiakban predikátum) kikapcsolható.

A predikátum lelke a `matches` és a `getMatcher` metódusok. Az utóbbi visszaad egy Clang-es AST matchert, amivel az előzetes mintaillesztést elvégezzük. Mivel csak a matcherekkel nem lehet egy node minden tulajdonságára mintát illeszteni, ezért miután a matcher elfogadott egy node-ot, a `matches` metódus leellenőrzi, hogy az adott node megfelel-e azoknak a követelménynek, amiket matcherekkel nem lehetett megfogalmazni. Ez a mechanizmus több *callback* segítségével működik. Az első hívás minden predikátum esetében azonos. Meghívja az adott predikátumhoz tartozó `matches` tagfüggvényt, ami gyakorlatilag így egy második *callback*. Ha az igaz értékkel tért vissza, akkor elvégzi a jelentést a felhasználó felé. Ebből a módszerből adódóan a jelentésért felelős kódrészlet nem került duplikálásra. A `getBoundID` metódus segítségével kapjuk meg azt az információt, hogy a matcherek által megjelölt node-ok közül melyiknek a forráskódbeli pozíciója szerepeljen a jelentésben. A dolgozat írásának idejében az implementáció HTML formátumban végezte a jelentést, egy összefoglaló statisztikával a végén.

A `MatcherProxy` létrehozására azért volt szükség, mert a matchereknek nincs közös ősosztálya, ami alapján polimorfikusan lennének kezelhetőek. A proxy osztályból kinyerni a matchert nehézkes, azonban erre csupán egy helyen van szükség. A módszer előnye, hogy elég egy `getMatcher` metódus a predikátumoknak, amire különben az eltérő visszatérési értékek miatt nem lehetne mód. Mivel a legtöbb matcher implicit módon konvertálódik a proxy osztállyá, ezért a metódusok maguk is kényelmesen kezelhetőek és bővíthetőek.

Elsősorban a mintaillesztés módját akarom bemutatni a különböző minták esetében. Ehhez minden egyes predikátum esetében elég a `matches` és a `getMatcher` metódusokat tárgyalni, mivel ezek határozzák meg, hogy az adott predikátum elfogad-e egy node-ot.

4.2. Predikátumok

4.2.1. Bool vector

A cél, hogy minden `vector<bool>` típusú változódeklarációt elkapjon az eszköz. Emiatt a változódeklarációkra akarunk mintát illeszteni. Érdekes belegondolni, hogy nem elég egy típust keresni. Jó példa erre egy `bool` mátrix, amit a következőképp is ábrázolhatunk: `vector<vector<bool>>`. Könnyű lenne azt vizsgálni, hogy az AST-ben szerepel-e valahol a `vector<bool>` típus, ugyanakkor ilyenkor nem lehet megmondani, hogy hol van az a deklaráció, ami hatására ez a sablon példányosodik. A dolgozat írásának idejében a matcherek egyik limitációja miatt nem volt lehetőség ennek a problémának az általános megoldására. Egy kerülő megoldásként ezért létrehoztam a `replicateTemplateSpecWithArgument` függvényt, ami egy típusra illeszt mintát az egymásba ágyazott sablon példányosítások között egy konfigurálható mélységig. Ez a mélység alapértelmezetten három, de szükség esetén egyszerűen módosítható. A `zeroLevel` paraméter megadja, hogy a paraméterként kapott típust elfogadjuk-e anélkül, hogy egy sablon paramétere lenne.

```
static DeclarationMatcher
    replicateTemplateSpecWithArgument (DeclarationMatcher decl,
                                       bool zeroLevel = false)
{
    auto templateSpecWithArgument = [] (DeclarationMatcher decl)
        -> DeclarationMatcher {
        return classTemplateSpecializationDecl(
            hasAnyTemplateArgument(
                refersToType(hasDeclaration(decl))));
    };

    DeclarationMatcher matcher = templateSpecWithArgument(decl);

    for(int i = 0; i < templateInstCheckNestDeph-1; ++i)
    {
        matcher = anyOf(matcher, templateSpecWithArgument(matcher));
    }

    if (zeroLevel)
        return anyOf(matcher, decl);
    else
        return matcher;
}
```

A *Tooling* könyvtár fejlesztői tudnak erről a limitációról, és dolgoznak már a megoldáson. Amint elérhető lesz, a kódokat refaktorálni fogom. A probléma ideiglenes megoldására szánt függvény viszont jól szemlélteti, hogy a matcherek egyszerűen generálhatóak akár futási időben is.

```

MatcherProxy StlBoolVectorPred::getMatcher()
{
    DeclarationMatcher decl =
        replicateTemplateSpecWithArgument(
            classTemplateSpecializationDecl(hasName("std::vector"),
                                           hasAnyTemplateArgument(
                                               refersToType(asString("_Bool")))), true);

    return varDecl(hasType(decl)).bind("id");
}

```

Mivel maga a minta egyszerű, amit keresünk, nem meglepő, hogy az apró kerülő megoldástól eltekintve a predikátum felépítése is egyszerű. Egy változó deklarációt keresünk, ami gyakorlatilag egy sablonspecializáció, ahol a sablon típusa `vector` és van `bool` típusú sablonparamétere. Ezt elfogadjuk önmagában, vagy egy másik sablon paramétereként. Mivel a változódeklaráció helyét akarjuk jelenteni a felhasználó felé, ezért hozzákötjük egy sztringhez, ami alapján a sok illeszkedő node-ból tudjuk, hogy melyiknek a forráskódbeli helyét akarjuk felhasználni a jelentéshez.

4.2.2. COAP

A `vector<bool>` kereséséhez képest nem nehezedik számottevően a feladat, ha tetszőleges konténerben akarunk `auto_ptr`-t keresni. Az előbbieken tárgyalt limitációk itt is érvényesek, és a megközelítés itt is azonos. Egyetlen eltérés, hogy itt futási időben állítunk elő egy matchert, ami az összes STL-ben megtalálható konténerre illeszkedik. Ehhez kézenfekvően adódik az `anyOf` matcher, aminek az egységelemként az `unless(anything())` használható, ami nem illeszkedik semmire. A `gContainers` egy olyan `vector<string>`, amiben az STL konténerek vannak felsorolva.

```

MatcherProxy StlCOAPPred::getMatcher()
{
    DeclarationMatcher type = unless(anything());
    for(const auto& e : gContainers)
        type = anyOf(recordDecl(hasName(e)), type);

    DeclarationMatcher decl =
        namedDecl(replicateTemplateSpecWithArgument(
            classTemplateSpecializationDecl(type,
                                           hasAnyTemplateArgument(
                                               refersToType(hasDeclaration(recordDecl(
                                                   hasName("std::auto_ptr"))))), true),
            unless(matchesName("iterator"))));

    return varDecl(hasType(decl)).bind("id");
}

```

Ami különbségként még megjelenik, az az `unless(matchesName("iterator"))`. Ennek egyik oka, hogy például a GCC STL implementációjában az iterátor egy `vector` esetében ilyen alakú:

```

__gnu_cxx::__normal_iterator<
    std::auto_ptr<int> *,
    std::vector<
        std::auto_ptr<int>,
        std::allocator<std::auto_ptr<int> >
    >
>
>

```

Sajnos az előbbi kikötés nélkül erre is illeszkedni fog a matcher, ami jogos, hiszen egyik template paraméter esetében megjelenik a COAP, ugyanakkor nem célszerű, hogy a felhasználó felé redundáns információ kerüljön jelentésre, hiszen azt a konténert, aminek az elemeire ez az iterátor mutatni fog, úgyis jelenti az eszköz.

4.2.3. Find és count

A feladat, hogy megtaláljuk azokat a `find` és `count` hívásokat, ahol az első paraméter típusa egy `set`hez vagy `multiset`hez tartozó `iterator` vagy `const_iterator`. Ez egy egyszerű feladatnak tűnhet, ugyanakkor ezek az iterátorok is egy-egy implementáció függő típus typedef-ei. Ez számos problémát okoz, ha STL implementációtól független módon akarjuk létrehozni a mintákat. Ennek a jobb megértésére tekintsük a `find(s.begin(), s.end(), 5)` kifejezésből generált AST egy részletét ahol az `s` egy `set<int>`.

```

(CallExpr 'find'
  (CXXConstructExpr '_Rb_tree_const_iterator'
    (MaterializeTemporaryExpr 'const_Rb_tree_const_iterator'
      (ImplicitCastExpr 'const_Rb_tree_const_iterator'
        (CXXMemberCallExpr 'iterator'
          (MemberExpr 'begin'
            (ImplicitCastExpr 'const'
              (DeclRefExpr 'set'))))))))

```

Ez az AST természetesen nem az eredeti Clang által generált AST, hanem annak egy szemléletesebb váza. Az említett kód első paraméterét láthatjuk. Az AST jól mutatja, hogy mi történik. A `set` implicit módon konvertálódik konstanssá, majd meghívódik a `begin` metódusa, ami vissza is ad nekünk egy iterátort, amiből létrejön egy temporális objektum egy konstruktorhíváson keresztül. Ami a problémát okozza, az a `CXXMemberCallExpr` node feletti implicit kasztolás. Ebben a kasztolásban elveszítjük azt az információt, hogy mi valójában egy iterátort kaptunk, így ha ezzel nem számolunk, nem fog illeszkedni a minta.

```

callExpr(callee(functionDecl(
  anyOf(hasName("std::find"), hasName("std::count"))),
  hasArgument(0,
  hasDescendant(expr(hasType(typedefType(hasDecl(
  matchesName(
    "std::(multiset|set).*::(const_iterator|iterator)"
  )))))))).bind("id");

```

Ezért nem elég megnézni, hogy a konstruktor milyen típusú objektumon hívódik meg az első paraméterben, hanem le kell ellenőrizni, hogy a konstruktor hívás gyerekei között megtalálható-e egy olyan típusra utaló node, aminek létezik olyan typedef-je, amire mintát akarunk illeszteni. Ezért felelős a **has-Descendant** és a belső matcherei. Ez a matcher nem fedi le azt az esetet, amikor az iterátor nem a paraméter helyén lesz megkonstruálva. Az az eset hasonlóan lekezelhető lenne, ugyanakkor a dolgozat írásának idejében egy limitáció miatt nem volt erre lehetőség.

4.2.4. Reverse iterator

A felhasználó figyelmeztetése a **reverse_iterator** base hívásának az esetleges veszélyeire egyszerű feladat, jól szemlélteti a matcherek erejét.

```
callExpr(callee(
  methodDecl(hasName("base"),
    ofClass(hasName("std::reverse_iterator")
    ))) .bind("id");
```

Egy olyan metódus hívást keresünk, aminek a neve **base** és az objektum, amin a hívás történt **reverse_iterator** típusú.

4.2.5. Funktor paraméterek típusa

A funktorok megkeresése szintén egyszerűen adódik. Azok az osztályok vagy struktúrák számítanak az ellenőrzés szempontjából relevánsnak, amik a **binary_function** vagy **unary_function** leszármazottai.

```
MatcherProxy StlBinFunctorTypeMismatchPred::getMatcher()
{
  return
  recordDecl(isDerivedFrom(classTemplateSpecializationDecl(
    hasTemplateArgument(0, refersToType(type().bind("T1"))),
    hasTemplateArgument(1, refersToType(type().bind("T2"))),
    hasTemplateArgument(2, refersToType(type().bind("T3"))),
    hasName("std::binary_function"))),
    hasDescendant(methodDecl(hasParameter(0,
      hasType(type().bind("P1")),
      hasParameter(1,
        hasType(type().bind("P2")),
        hasName("operator()"),
        returns(type().bind("P3"))
      ))) .bind("id");
}
```

A matcherek nem alkalmasak arra, hogy két típust összehasonlítsanak, ezért az egyetlen lehetőség, hogy ezt a lépést később egy *callback* végezze el. Ehhez azonban meg kell jelölni azokat a node-okat, amikkel a *callback* dolgozni fog. Ezek a **binary_function** sablon típusparaméterei, valamint a leszármazottjának az **operator()** metódusának a paramétereinek a típusai, valamint a visszatérési érték típusa.

```

bool StlBinFuntorTypeMismatchPred::matches
    (const ast_matchers::MatchFinder::MatchResult &result_)
{
    const Type* T1 = result_.Nodes.getDeclAs<Type>("T1");
    const Type* T2 = result_.Nodes.getDeclAs<Type>("T2");
    const Type* T3 = result_.Nodes.getDeclAs<Type>("T3");
    const Type* P1 = result_.Nodes.getDeclAs<Type>("P1");
    const Type* P2 = result_.Nodes.getDeclAs<Type>("P2");
    const Type* P3 = result_.Nodes.getDeclAs<Type>("P3");

    assert(T1 && T2 && T3 && P1 && P2 && P3);

    return ((T1->getCanonicalTypeUnqualified() !=
              P1->getCanonicalTypeUnqualified()) ||
            (T2->getCanonicalTypeUnqualified() !=
              P2->getCanonicalTypeUnqualified()) ||
            (T3->getCanonicalTypeUnqualified() !=
              P3->getCanonicalTypeUnqualified()));
}

```

Az összehasonlítást elvégző *callback* felépítése nagyon egyszerű. Miután megkapta a találatához tartozó node-okat, csupán pár egyszerű összehasonlításra van szükség.

Az `unary_function` esetében is hasonlóan járunk el, sajnos a két eset nem vonható össze, de a gondolatmenet megegyezik.

Ennek a predikátumnak az implementációja kapcsán érdemes megfontolni az AST-n való keresés előnyeit. A minta egyszerű, viszont nagyon szemléletes következtetéseket lehet levonni. A makrók számos nehézséget okozhatnak szöveg alapú analízis esetén. Az általuk kigenerált kódok ellenőrzése bonyolult, a legbiztosabb módszer, ha a már előfeldolgozott fájlokon végezzük az ellenőrzést. Ezért előnyös, hogy az AST mindig az előfeldolgozott fordítási egységekből jön létre. Egy másik felmerülő probléma a `typedef`ekkel kapcsolatos. Főleg, mivel tetszőlegesen bonyolult lánc vagy akár háló építhető fel belőlük. Érdemes megfontolni azokat az eseteket is, amikor a funktor nem közvetlen származik le a számunkra érdekes osztályokból. Ezek a problémák mind egyszerűen kezelhetővé válnak, amint az AST-n végezzük a keresést.

4.2.6. Állapottal rendelkező allokátorok

A matcher felépítése majdnem megegyezik a COAP-nál felépítéssel, azzal a különbséggel, hogy nincs megkötés a sablonok típusparamétereire. Ennek az oka, hogy bár létezik matcher, amivel feltételek szabhatóak meg egy adott indexű paraméterre, de nem lehet feltételt adni az utolsóra. A konténerek eltérő mennyiségű sablon paramétert várnak, és minden esetben az utolsó az allokátor. Az allokátor meghatározása ezért egy *callback* feladata.

```

MatcherProxy StlAllocWithStatePred::getMatcher()
{
    DeclarationMatcher container = unless(anything());
    for(const auto& e : gContainers)
        container = anyOf(recordDecl(hasName(e)), container);

    return
        varDecl(hasType(namedDecl(
            replicateTemplateSpecWithArgument(
                allOf(classTemplateSpecializationDecl(),
                    container), true),
            unless(matchesName("iterator"))
                .bind("alloc"))).bind("id"));
}

```

A *callback* felépítése egyszerű. Egyetlen feladata meghatározni, hogy az utolsó template paraméter egy osztály-e, amennyiben igen, rendelkezik-e legalább egy adattaggal.

```

bool StlAllocWithStatePred::matches
    (const ast_matchers::MatchFinder::MatchResult &result_)
{
    const ClassTemplateSpecializationDecl* D =
        result_.Nodes.getDeclAs<ClassTemplateSpecializationDecl>("alloc");

    const auto& list = D->getTemplateArgs();
    const auto& arg = list.get(list.size()-1);

    assert(arg.getKind() == TemplateArgument::Type);
    const auto& type = arg.getAsType();

    const CXXRecordDecl* decl = type->getAsCXXRecordDecl();

    if (!decl) return false;

    return !decl->field_empty();
}

```

4.2.7. Állapottal rendelkező predikátumok

Azon funktorok kiszűrése, amiknek az eredménye egy állapottól is függ, egyszerű feladat. A problémát az okozza, hogy eldöntsük egy funktorról, hogy predikátum-e. Általában azok a funktorok predikátumok, amit olyan algoritmusnak adunk át, amik predikátumot várnak. Ezért kilistázhathatnánk ezeket a standard algoritmusokat, ekkor azonban nem működne az ellenőrzés a felhasználók által írt algoritmusokra. Ezért az ellenőrzéskor azokat a funktorokat tekintem predikátumoknak, amelyeknél az `operator()` visszatérési értéke `bool`. Ez nem fed le azokat a predikátumokat, ahol a visszatérési érték implicit módon konvertálódik `bool`-ra. Emellett a `bool` visszatérési érték nem feltétlen jelenti azt, hogy az adott funktor predikátumként lesz felhasználva.

```
thisExpr(hasAncestor(
  methodDecl(hasName("operator()"),
    returns(asString("_Bool")))).bind("id");
```

A minta maga egyszerű. Az AST-ben, ha egy osztály tagjára hivatkozunk, még ha nem is írjuk ki, implicit módon megjelenik egy `this` kifejezés. Ez arra utal, hogy használjuk az osztály egyik adattagját. Kiindulva egy ilyen kifejezésből könnyen leellenőrizhető, hogy milyen függvényben szerepel. Ha ez egy `operator()`, aminek a visszatérési értéke `bool`, akkor a minta elfogadja a kifejezést. Ennek a megközelítésnek nagy előnye, hogy a felhasználó számára közvetlen az a kifejezés kerül jelentésre, ami miatt nem független a predikátum az állapotoktól.

4.2.8. Polimorfikus funktorok

Predikátumokhoz hasonló probléma definiálni, hogy mit tekintünk funktornak. A legkézenfekvőbb megoldás erre, ha azt az osztályt vesszük annak, ami rendelkezik `operator()` metódussal. Ezek keresésére rövid minta illeszthető: `methodDecl(hasName("operator()")).bind("id")`. Az egyetlen probléma, hogy nincs matcher arra, hogy meghatározzuk egy metódusról, hogy *virtual*-e. Ez a feladat így megint egy *callback*-re marad.

```
bool StlPolimorphFunctorPred::matches
  (const ast_matchers::MatchFinder::MatchResult &result_)
{
  const CXXMethodDecl* D =
    result_.Nodes.getDeclAs<CXXMethodDecl>("id");

  return (D && !D->isVirtual());
}
```

Itt érdemes megemlíteni, hogy szöveg alapú statikus analízisnél újabb problémák adódnának. Egy metódus nem csak akkor lehet *virtual*, ha ez explicit módon ki van elé írva, elég hozzá, ha az egyik őszülő rendelkezik azonos nevű és paraméterezésű metódussal, ami *virtual*ként van deklarálva. Az AST-n való keresés esetén ez természetesen nem jelent problémát.

4.2.9. Ürességvizsgálat

A cél, hogy meghatározzuk azokat a helyeket, ahol egy konténer ürességét a `size` metódussal vizsgálják. Számos eset lehetséges, ezeket mind le kell fedni.

```

MatcherProxy StlEmptyCheckPred::getMatcher()
{
    DeclarationMatcher container = unless(anything());
    for(const auto& e : gContainers)
        container = anyOf(recordDecl(hasName(e)), container);

    return
        memberCallExpr(on(hasType(container)),
            callee(methodDecl(hasName("size"))),
            anyOf(
                unless(anyOf(hasParent(binaryOperator(
                    unless(has(integerLiteral(equals(0))))),
                    hasAncestor(varDecl()),
                    hasAncestor(binaryOperator(
                        hasOperatorName("="))))),
                    hasParent(implicitCastExpr(
                        hasImplicitDestinationType(
                            asString("_Bool")))))
                ).bind("id");
            }
}

```

A predikátum a következő eseteket fedi le:

- A **size** metódus eredménye a 0 konstansal van összehasonlítva, akkor az ürességvizsgálat. Ezt egy olyan minta fedi le, ami akkor fogadja el a metódushívást, ha annak szülője a szintaxisfában egy bináris operátor és az operátor másik oldalán egy 0 konstans áll.
- A **size** metódus hívása önmagában áll egy logikai kifejezésben és nincs összehasonlítva semmilyen konstansal vagy változóval. Ez esetben a visszatérési értéke implicit módon konvertálódik logikai értéké.
- A **size** metódus visszatérési értékével történő inicializáció, az csak abban az esetben ürességvizsgálat, ha a változó típusa **bool**. Ekkor is megjelenik egy implicit konverzió.
- A **size** metódus visszatérési értékét nem **bool** változónak adjuk értékül, akkor az nem számít ürességvizsgálatnak, így nem kerül detektálásra, ez kivételként szerepel a mintában. **Bool** változónak való értékadás esetén szintén megjelenik egy implicit konverzió.
- A **!** unáris operátor mögött szereplő **size** hívás esetén is megjelenik egy implicit konverzió.

4.2.10. Shrink to fit

Ha egy konténer kapacitását le akarjuk csökkenteni, hogy pontosan az elemek számával egyezzen meg, azt a következőképp tehetjük meg: `container<T>(c).swap(c)`. Erre a C++11 szabvány egy hatékonyabb megoldást kínál, a `shrink_to_fit` metódust. Ezért a meglévő kódok migrálása szempontjából

hasznos, ha egy eszköz megtalálja az összes ilyen kódrészletet. A minta megértése érdekében tekintsük a `vector<int>(v).swap(v)` kódrészlet által generált AST-t.

```
(ExprWithCleanups
 (CXXMemberCallExpr
  (MemberExpr 'swap'
   (CXXFunctionalCastExpr
    (CXXBindTemporaryExpr 'std::vector<int>'
     (CXXConstructExpr 'vector<int>'
      (ImplicitCastExpr
       (DeclRefExpr 'vector<int>_v')))))
    (DeclRefExpr 'vector<int>_v'))))
 (DeclRefExpr 'vector<int>_v')))
```

Egy éppen megkonstruált temporális objektumról és a rajta végzett konstruktor hívásról van szó, aminek a másoló konstruktora ugyanazt az objektumot kapja paraméterül, mint a rajta hívott swap. A minta is ezt írja le néhány csavarral. Az egyik probléma, hogy matcherekkel nem lehet megállapítani, hogy a konstruktor és a swap paramétere megegyezik-e, ezért ezt megint egy *callback* fogja meghatározni. A másik gond, hogy a paraméter nem mindig `DeclRefExpr`-ként jelenik meg az AST-ben. Amennyiben egy rekord tagja a paraméter, ez esetben `memberExpr` lesz az AST-ben.

```
MatcherProxy StlShrinkPred::getMatcher()
{
    const auto shrinkables =
        {"std::deque", "std::basic_string", "std::vector"};
    DeclarationMatcher container = unless(anything());
    for(const auto& e : shrinkables)
        container = anyOf(recordDecl(hasName(e)), container);

    return
        memberCallExpr(on(hasType(container)),
            callee(methodDecl(hasName("swap"))),
            has(memberExpr(hasDescendant(constructExpr(
                hasArgument(0, anyOf(memberExpr().bind("CP1"),
                    declRefExpr().bind("CP2"))))))),
                hasArgument(0, anyOf(memberExpr().bind("SP1"),
                    declRefExpr().bind("SP2")))).bind("id");
}
```

A *callback*-ben meghatározzuk, hogy melyik deklaráció tartozik a kapott objektumhoz. Ha az ezekhez tartozó forráskódbeli pozíció ugyanaz, akkor a deklaráció is ugyanaz, tehát illeszkedni fog a minta.

4.2.11. Remove-val kapcsolatos hibák

Az első egyértelmű hiba, amikor a `remove`, `remove_if` vagy `unique` függvények visszatérési értéke nem kerül felhasználásra. Ebben az esetben a hívás értelmetlen volt, törlés valójában nem történik. A visszatérési érték felhasználásának módja többféle lehet. Szerepelhet egy függvényhívás paramétereként, értékadásként vagy konstruktor paramétereként. Természetesen az

nem jelent semmiféle garanciát arra, hogy az elemek valóban törölve lesznek, ha a visszatérési érték eltárolásra került. Viszont, ha a visszatérési érték semmilyen módon nem lett felhasználva, az biztos hibát jelent. Ezt az eszköz egy figyelmeztetéssel jelzi.

```
callExpr(callee(functionDecl(anyOf(matchesName("std::remove"),
                                   hasName("std::unique")))),
         unless(anyOf(hasAncestor(operatorCallExpr(
             hasOverloadedOperatorName("="))),
                     hasAncestor(constructExpr()),
                     hasAncestor(callExpr()))).bind("id");
```

A `remove`-ra és `remove_if`-re egyszerre is lehet mintát illeszteni a `matchesName` matcher segítségével, ami paraméterként reguláris kifejezést vár. Mivel az `std::remove` kifejezés elfogadja a `std::remove_if`-et is, ezért a minta arra is illeszkedni fog.

A második felmerülő probléma, amikor ugyan átadjuk az eredményt az `erase` metódusnak, de csupán ezt az egy iterátort. Ennek a kiszűrése abban az esetben egyszerű, amikor például a `remove` függvény hívása paraméterként szerepel. Ekkor ha nincs más paraméter, akkor a minta elfogadja az AST részletet.

```
callExpr(hasDeclaration(methodDecl(hasName("erase"))),
         argumentCountIs(1),
         hasArgument(0, hasDescendant(callExpr(callee(
             functionDecl(anyOf(matchesName("std::remove"),
                                   hasName("std::unique"))))))).bind("id");
```

Abban az esetben, amikor az `erase` egy iterátort kap, ami előtte lett beállítva egy `remove` hívás visszatérési értékére, elsősorban az a limitáció korlátoz, miszerint jelenleg nincsenek olyan matcherek, amelyek forráskódbeli pozícióra adnak megszorítást. Ez a probléma számos alkalommal előjön, ha sikerül ezeket a matchereket implementálni, az nagy előrelépést fog jelenteni a leírható mintákban. *Callback*-kel lehetséges kerülőmegoldást létrehozni, de az bonyolult, és a teljesítménye sem megfelelő.

4.2.12. Másoló algoritmusok és iterátor adapterek

Másoló algoritmusok esetén minden olyan esetben létre kell, hogy jöjjön egy figyelmeztetés, amikor az utolsó paraméter nem egy iterátor adapter. Mivel minden másoló STL függvény nevében a `transform`ot kivéve megtalálható a `copy` szó, ezért könnyű rájuk reguláris kifejezést megadni.

```
callExpr(callee(functionDecl(matchesName(
    "std::(*copy.*|transform)")),
         hasArgument(2, hasDescendant(callExpr(
             callee(functionDecl(matchesName(
                 "std::.*inserter.*"))))))).bind("id");
```

Az iterátor adapterek esetében is szerencsés a helyzet, mivel ezek nevében az `inserter` fordul elő minden esetben. Ezen tulajdonságok miatt a minta kompakt módon leírható.

4.2.13. Polimorfikusan használt konténerek

Egy `vector<int>* ptr = new Derived;` hívás esetén egy implicit konverzió jön létre, ahol a leszármazott osztályra mutató mutató kasztolódik bázisosztályra mutató mutatóvá.

```
MatcherProxy StlPolimoContPred::getMatcher()
{
    DeclarationMatcher container = unless(anything());
    for(const auto& e : gContainers)
        container = anyOf(recordDecl(hasName(e)), container);

    return
        implicitCastExpr(hasImplicitDestinationType(
            pointsTo(container))).bind("id");
}
```

A minta ezért azokra az AST részletekre illeszkedik, ahol egy olyan implicit kasztolás jelenik meg, ahol a céltípus egy konténerre mutató mutató. Azonban azt a matcherek segítségével nem lehet megállapítani, hogy a konverzió típusa milyen jellegű.

```
bool StlPolimoContPred::matches
    (const ast_matchers::MatchFinder::MatchResult &result_)
{
    const ImplicitCastExpr* cast =
        result_.Nodes.getDeclAs<ImplicitCastExpr>("id");

    return cast->getCastKind () == CK_DerivedToBase;
}
```

Ezért a predikátumhoz tartozó *callback* leellenőrzi, hogy a konverzió valóban megfelel-e a követelményeknek. Ezt az információt az AST-ből könnyen kinyerhetjük. A minta akkor is illeszkedik, ha `auto_ptr` típusparamétereként szerepel STL konténer, mivel ekkor az `auto_ptr` konstruktora a konténerre mutató pointert vár, ezért itt is létrejön az implicit konverzió.

4.2.14. Tiltott temporálisok

Szinkronizációs primitívekből nem ajánlott temporális objektum létrehozása, ezért ahol ezek létrejönnek, oda az eszköz figyelmeztetést fog generálni. Az, hogy ezekből a temporális objektumokból milyen károk származhatnak, nem tartozik szervesen a dolgozat témájához, így ezt nem részletezem.

```

MatcherProxy StlForbiddenTemporaryPred::getMatcher()
{
    const auto syncTypes = {"std::mutex", "std::timed_mutex",
        "std::recursive_mutex", "std::recursive_timed_mutex",
        "std::lock_guard", "std::unique_lock"};

    DeclarationMatcher syncs = unless(anything());
    for(const auto& e : syncTypes)
        syncs = anyOf(recordDecl(hasName(e)), syncs);

    return materializeTemporaryExpr(hasType(syncs)).bind("id");
}

```

A minta a temporálisan létrejövő objektumokat a típusuk alapján fogadja el. Az AST szerkezete következtében a matcher felépítése egyszerű.

5. fejezet

Távlatok

5.1. Implementációt érintő fejlesztések

A matcherek könnyen komponálhatóak az egymásba ágyazásukkal. Érdemes az általam kialakított predikátumokat is komponálhatóvá tenni, ezáltal általánosabban paraméterezhető minták is megfogalmazhatóak magasabb absztrakciós szinten.

A Clang fordító C++ mellett támogatja a C, Obj-C és Obj-C++ nyelveket is. Ebből adódóan az implementációt egyszerűen ki lehet egészíteni olyan ellenőrzésekkel, amik akár mindegyik támogatott nyelv esetén relevánsak, vagy akár egy másik támogatott nyelvre specifikusak.

A jelenlegi implementáció futása teljes mértékben szekvenciális. Ugyanakkor az egyes fordítási egységek feldolgozásai nincsenek egymásra semmilyen hatással, egymástól teljesen függetlenek. Ebből adódóan ez a folyamat könnyedén párhuzamosítható. Mivel a feldolgozást a Clang fordító végzi, ezért ez a fejlesztés elsősorban Clang fejlesztés lenne.

A LibTooling könyvtár még nem támogatja, hogy ne minden fordítási egység kerüljön feldolgozásra, csupán néhány. Ez a támogatás elsősorban a felhasználók szempontjából lenne előnyös, mivel egy nagy projekt esetén hosszadalmas idő, ameddig lefut a statikus analízis. Ugyanakkor egy-egy fordítási egység feldolgozása másodperceken belül elkészül. Ezért az eszköz valós idejű ellenőrzésre csak akkor lesz alkalmas, ha ez a funkcionalitás implementálásra kerül.

A dolgozat írásának az idejében az ASTMatcher könyvtár nem biztosított lehetőséget arra, hogy ne teljes AST, hanem csak annak egy részfája kerüljön feldolgozásra. Egy-egy részfa feldolgozása akár a másodperc törtrésze alatt is megtörténhet, ezért ebben az esetben az eszköz akár egy IDE-ben az

intellisense-szel együttműködve szemantikus analízist hajthat végre valós időben.

Jelenleg minden figyelmeztetés esetében csupán egy helyet jelöl az eszköz a forrásban. Egy lehetséges továbbfejlesztése a felhasználó értesítésének, ha a vonatkozó egyéb pozíciókat is megjelöljük, például nem csupán egy problémás függvény hívásának a helyét, hanem a definícióját is.

5.2. Az elmélet továbbfejlesztése

A dolgozat írásának az idejében a felhasznált technológiák egy része bár jól használható volt, de még gyerekcipőben járt. Ezt jól jelzik a limitációk, amelyek gyakran azt eredményezték, hogy az elméletet kellett az implementációhoz igazítani és nem fordítva. A technológia fejlődésével érdemes lesz átdolgozni az eddig elért eredmények egy részét.

Az itt bemutatott ellenőrzések mellett még számos egyéb *match* készíthető. Jó lenne például iterátorok invalidálódását detektálni. Nem csupán az STL-hez kapcsolódó ellenőrzések jönnek szóba, a tipikus C++ hibák mellett a C++11-es szabvány buktatóit is lehetséges ellenőrizni. A szabványok közti migrálást megkönnyítő minták száma és bonyolultsága egy jó mértéke lehet a visszafelé kompatibilitásnak.

Amennyiben majd sikerült elég mennyiségű szerteágazó funkciójú mintát összegyűjteni, érdemes őket általánosítani, tulajdonságaikat vizsgálni, ezekből következtetéseket levonni. A gyakran ismétlődő részeket ki kell emelni magasabb szintű paraméterezhető mintákká, amikkel később akár szemantikailag bonyolultabb összefüggések is vizsgálhatóak. Ezekből a mintákból definiálható egy lekérdező nyelv, aminek a használata nagy segítség lehet hibák keresésénél vagy akár a kód megértésénél is.

Mivel minden modern nyelv rendelkezik valamilyen standard könyvtár készlettel, ezért a leszűrt tapasztalatok átemelhetőek akár más nyelvi környezetbe is.

A dolgozat elsősorban a szemantikus ellenőrzésekre tért ki, de a mintaillesztés felhasználható refaktorálási folyamatokban is, ahol a legnagyobb kihívást az érintett kódrészlet megkeresése jelenti. Emellett eszközt ad újabb kódmetrikák meghatározására és implementálására.

6. fejezet

Összefoglalás

Az STL felhasználásának az elemzésén keresztül bemutattam egy új megközelítést a C++ kódok statikus elemzésére. A megközelítés lényege, hogy predikátumok segítségével illesztünk mintát az AST-n, ahol a predikátumok kombinálhatóak és egy beágyazott nyelvet hoznak létre. A teljes AST lefedéséhez nagyon összetett nyelvre lenne szükség, nem várható el, hogy a predikátumokkal tetszőleges minta leírható legyen. Eredetileg ezért ezt a nyelvet úgy definiálták, hogy API refaktorálási célokra könnyen felhasználható legyen. Arra a következtetésre jutottam, hogy elég egy kezdeti szűkítést leírni a találatokra, amit később egy callbackben lehet finomítani, ahol már teljes a hozzáférés az AST-hez. A callback-et és a kezdeti szűkítést végző mintát összevontam, ezáltal egy magasabb szintű mintát kaptam, amit predikátumként kezeltem.

A módszer hatékonyságát és használhatóságát az általam létrehozott implementáció igazolja. Ez az eszköz az elsők között volt, ami felhasználta az ASTMatcher technológiát. A munka során folyamatos kapcsolatban álltam a Google azon alkalmazottaival, akik a matcherek fejlesztésével foglalkoznak. Az implementáció fejlesztése folyamán számos hibára és limitációra derítettem fényt a matcherekkel kapcsolatban. Emellett amikor egy-egy ellenőrzés következtében felmerült az igény még nem létező matcherekre, azokat implementáltam, és ezek bekerültek a Clang kódbázisába¹.

Az implementációhoz sikerült egy előnyös felépítést találnom, aminek a következtében gyorsan és könnyen bővíthető új ellenőrzésekkel, anélkül, hogy a program többi részét ismerni kellene. Az eszköz egy módosított változatát az ipari partnerünk, az Ericsson is felhasználja. A módosítások elsősorban további predikátumokat jelentenek, amik a cég kódbázisára specifikusak. Az architektúra sikerességét bizonyítja, hogy ezeket a kiegészítéseket az Ericsson

¹Az általam beküldött módosítások: r164943, r164304, r166477, r164123 [17].

son Hungary dolgozói el tudták végezni az implementáció részletes ismerete nélkül.

Irodalomjegyzék

- [1] Austern, M. H.: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley (1998)
- [2] Dévai, G., Pataki, N.: *Towards verified usage of the C++ Standard Template Library*, In Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST) 2007, pp. 360–371.
- [3] Gregor, D., Schupp, S.: *STLlint: Lifting static checking from languages to libraries*, Software: Practice and Experience **36(3)** (2006), pp. 225–254.
- [4] Karpov, A.: Errors detected in the Visual C++ 2012 libraries, <http://www.viva64.com/en/b/0163/> (2012. okt. 20)
- [5] Lattner, C.: *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.
- [6] Meyers, S.: *More Effective C++ – 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley (1995)
- [7] Meyers, S.: *Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley (2001)
- [8] Meyers, S.: *Effective C++ – 55 Specific Ways to Improve Your Programs and Designs* (3rd Edition), Addison-Wesley (2005)
- [9] Pataki, N.: *Compile-time Advances of the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **36** (2012), Selected papers of 9th Joint Conference on Mathematics and Computer Science MaCS 2012, pp. 341–353.
- [10] Pataki, N., Szűgyi, Z., Dévai, G.: *C++ Standard Template Library in a Safer Way*, In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.

- [11] Paul, S., Prakash, A.: *A Framework for Source Code Search using Program Patterns*, IEEE Transactions on Software Engineering **20** (1994), pp. 463–475.
- [12] Stroustrup, B.: *A C++ programozási nyelv*, Kiskapu Kiadó (2000)
- [13] Clang: a C language family frontend for LLVM, <http://clang.llvm.org/> (2012. okt. 20)
- [14] Clang LibTooling, <http://clang.llvm.org/docs/LibTooling.html> (2012. okt. 20)
- [15] CppDepend, <http://www.cppdepend.com/> (2012. okt. 20)
- [16] Google C++ Mocking Framework, <http://code.google.com/p/googlemock/> (2012. okt. 20)
- [17] LLVM/Clang SVN History, <http://llvm.org/viewvc/llvm-project/cfe/trunk/> (2012. okt. 20)