Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Somogyi Norbert Zsolt

FORRÁSKÓD MODERNIZÁCIÓJA MODELL ALAPÚ MEGKÖZELÍTÉSSEL

KÖVESDÁN GÁBOR

BUDAPEST, 2019

Tartalomjegyzék

Összefoglaló

Az elavult szoftverek modern környezetbe történő átültetése mindig bonyolult feladat. Sok ilyen szoftver még mindig használatban van és a korszerűtlen tervezési megfontolások és elavult technológiai megoldások miatt karbantartásuk költséges, és a lecserélésük nehéz. Ezeket a szoftvereket úgy kívánatos modern környezetbe átültetni, hogy a korszerű szoftvertervezési konvenciókat kövessék, tehát jól olvasható és karbantartható kódjuk legyen, és a kompatibilitást is teljesen megőrizzék. A rendelkezésre állási követelmények miatt általában nem lehet az eredeti rendszert leállítani, és az új verziót éles környezetben tesztelni.

Ezeket a nehézségeket könnyítik meg a különböző kódmodernizációs eszközök, melyek elavult szemléletben és nyelven íródott forráskódot ültetnek át korszerű és előnyös környezetbe. Ennek legcélszerűbb módja a jellemzően procedurális kód átalakítása objektumorientált paradigmákat követő szoftverré. Ezért az elavult kódbázis egyszerű átültetése egy modern nyelvre nem elégséges megoldás. A probléma egyik legnagyobb nehézsége egy előnyös objektumorientált szerkezet kialakítása úgy, hogy az eredeti alkalmazás szemantikája ne változzon meg. További nehézség, hogy mindezt automatikusan kell megoldani. Bár a szoftvermodernizáció szakértői szerint a teljeskörű, helyes kimenetet generáló automatizálás megoldhatatlan feladat, törekedni kell a minél nagyobb fokú automatizálás kialakítására. A kódmodernizáció harmadik jelentős kihívása az eszközök teljesítménye, vagyis hogy milyen jellemző memóriaigénnyel és mennyi idő alatt képesek modernizálni komplex, többszázezer sorból álló alkalmazásokat.

Az általam javasolt megoldásban ismertetek egy általános módszertant, amely procedurális szemléletben íródott szoftvereket képez le ekvivalens szemantikával rendelkező objektumorientált szerkezetre. A megoldásom újszerűségét a modell-alapú megközelítése adja. A modell-vezérelt módszerek és eszközök terén az utóbbi években nagy fejlődést tapasztalhattunk, ezért újszerű megoldás a problémakör modell-vezérelt szemléletű vizsgálata. Ennek egyik szembetűnő előnye, hogy a transzformáció jelentős része nyelvfüggetlen módon fog tudni működni, ami megkönnyíti a transzformátor különböző forrásnyelvekre és célnyelvekre való megvalósítását.

A kidolgozott módszertan mellett bemutatok egy általam készített transpilert is, mely a módszertant valósítja meg a gyakorlatban. Ennek segítségével szemléltetem a módszerem működőképességét. Mivel a teljesítmény kritikus kérdés a kódmodernizáció területén, ezért komplex nyílt-forráskódú szoftverek modernizálásának esettanulmányán keresztül bizonyítom a módszertan és az elkészített eszköz hatékonyságát is.

Abstract

Modernizing legacy software is always a difficult task. Many of these software are still in use and because of their outdated design their maintenance is expensive and are hard to replace. Transplanting them into a modern environment should be done in such a way that makes them follow modern software design conventions resulting in well readable and maintainable code while preserving full compatibility with the original system. There are often strict requirements of availability that make it impossible to shut down the system and test it in the new environment.

These difficulties can be easened using different code modernization tools, which transplant legacy code into a modern and advantageous environment. The most preferable approach of this is to convert the often procedural legacy code into an object oriented software. Because of this simply rewriting the original code in a modern language is not a sufficient solution. One of the main difficulties of code modernization is to create an efficient object oriented structure from the original code without changing its semantics. Moreover, all this has to be done automatically. While the main experts of code modernization mostly agree that generating a completely error-free solution automatically is an impossible task, the process should be as automatic as possible. The third main challenge of this field is the performance of code modernizing tools, meaning mostly their average memory consumption and runtime while modernizing complex applications that may consist of hundred-thousands of lines of code.

In my solution I propose a general methodology that is capable of transforming procedural code into an equivalent object oriented software. The main strength of my solution is its modell-based approach. In recent years there have been huge improvements in the field of modell-driven methods and tools, making the modell-driven approach a promising and new way of modernizing legacy code. A basic advantage of this approach is that a significant part of the transformation methodologies will be able to be language independent, making it easier to implement a transpiler for different source and target languages.

I will also present a transpiler created by me that implements my methodology in practice. Using this I will illustrate the operability of my methodology. Because performance is a very critical part of code modernization, I will also present case studies of modernizing complex open-source software with my transpiler, thus proving the efficiency of my solution.

Bevezetés

1.1 Előzmények

Jelen dolgozat folytatása és továbbfejlesztése a 2017-ben készült TDK dolgozatomnak, amely a következő linken érhető el:

http://tdk.bme.hu/VIK/SW4/Forraskod-transzformacioja-Eclipse-kornyezetben

Az előző dolgozathoz képest a következő változások történtek:

- A módszertant bővítem az ***5.3*** fejezetben ismertetett **API hívások transzformációjával**. A módszertan többi része az előző dolgozatomban is szerepel.
- Az előző dolgozatban a **transpiler** egyáltalán nem foglalkozott utasításokkal, csak a program szerkezetét transzformálta. Ez azt jelenti, hogy a procedurális kód objektumorientált struktúrába szervezésén dolgoztam, és a metódusok törzsei kitöltetlenek maradtak. Mostanra az eszközt a ***6.5*** fejezetben leírt hiányossággoktól eltekintve elkészítettem. Ez azért is fontos, mert most már teljesértékű skálázhatósági vizsgálatok végezhetők. Legutóbb ez nem így volt, mert az alkalmazás transzformálásának nagy része hiányzott.
- A transpiler elkészítését **szisztematikus funkcionális teszteléssel** támogatom.
- Az eszközön **skálázhatósági vizsgálatokat** is végzek valós, komplex szoftvereken.
- A probléma **irodalomkutatása** az előző dolgozathoz képest alaposabb és részletesebb lett és alaposabban tárgyalom azt is, hogy a megoldásom ezektől miben tér el, vagy miben alkalmaz újszerű megközelítést. Megvizsgáltam továbbá számos kódelemző eszközt is.
- Nyílt forráskódú szoftverek **esettanulmányain keresztül** vizsgálom a transpiler működését.

1.2 Motiváció

A szoftvermodernizáció területe az elavult szoftverek modern környezetbe való átültetésével foglalkozik. Ezek a szoftverek, elavult tervezési szerkezetük miatt, általában nehezen karbantarthatók és áttekinthetők. Ezért ezeket célszerű modern környezetben futó, átlátható és könnyebben kezelhető rendszerekké átalakítani. Ennek legcélravezetőbb módja az, hogy az eredeti kódot azonos szemantikával rendelkező, de objektumorientált szerkezetű kódra alakítjuk át. Ez a transzformáció azonban nehéz feladat: több olyan fontos szempont van, amelyek kritikus pontjai egy ilyen átalakításnak. A transzformáció minél nagyobb fokú automatizálását támogatni kell annak ellenére, hogy annak teljes automatizálása a szoftvermodernizáció területének szakértői szerint lehetetlen feladat. Fontos szempont továbbá az is, hogy a transzformáció hatékony legyen futásidő és memóriahasználat szempontjából.

A kódmodernizációs eszközök feladata ezeknek a problémáknak a megkönnyítése. Megjegyzendő, hogy az eszközök futtatása után általában szükség van további kézi refaktorálási műveletekre. Az elkészült kódon célszerű futtatni az adott platformra készült refaktoráló eszközt is, amely növelheti a generált kód hatékonyságát modern nyelvi elemek használatával. Szükséges továbbá az eredeti platformon használt külső függőségek lecserélése az új platform megfelelő keretrendszereivel. Ha ezek forráskódja nem elérhető (és általában ez a helyzet), akkor ez természetesen érdemben nem automatizálható folyamat. Ezért az új keretrendszerek rendeltetésszerű használata a programozó feladata marad.

1.3 Procedurális és objektumorientált felépítés

A legacy szoftverek jellemzően procedurális szemléletben íródtak. Ennek lényege, hogy definiáljuk a használandó adatstruktúrákat, és a programot funkcionális dekompozícióval globális függvényekre bontjuk, amelyek az adatstruktúrákon műveleteket végeznek. Egyes függvények logikailag tartozhatnak ugyan bizonyos adatstruktúrákhoz, de nyelvi szinten az adatok és a rajtuk műveletet végző kód nincsenek egységbe zárva. Ez eltér az emberi szemlélettől, amelyben a programban szereplő fogalmakra (entitásokra) a hozzájuk kapcsolódó műveletekkel együtt gondolunk. Másrészt, nehéz a felelősségek elkülönítése (separation of concerns), és a globális állapotok és függvények használata nehezen átláthatóvá teszi a programkódot.

Ezzel szemben az objektumorientált felépítést követő programok az objektum fogalmának segítségével közelebb állnak az emberi gondolkodáshoz. A négy alap paradigma – egységbezárás, polimorfizmus, öröklés, absztrakció – betartásával a felelősségek jobban elkülöníthetők, és az egyes komponensek könnyebben újrafelhasználhatók, jobban olvashatók és egyszerűbben karbantarthatók lehetnek.

Az objektumorientált programozási nyelvek tehát a modern szoftverekben egyértelműen preferenciát élveznek a procedurális nyelvekkel szemben.

1.4 Célkitűzés

Dolgozatom célja a szoftvermodernizáció területének modell-vezérelt vizsgálata. Ennek keretein belül bemutatok egy általam kidolgozott módszertant, amely procedurális felépítésű kódot képes objektumorientált szerkezetű szoftverré átalakítani.

Ahogy korábban említésre került, a transzformáció teljeskörű automatizálása nem megoldható feladat. Adható azonban félautomata megoldás, amely „nagyjából" helyes kódot generál. A generált kód nem lesz hibamentes. A cél, hogy az eszköz a modernizálást nagyban megkönnyítse azáltal, hogy „elég jó" kódot generál, amit kézi refaktorálási lépésekkel már sokkal könnyebb és gyorsabb kijavítani. Munkám célja ennek megfelelően félautomata megoldást adni a kódmodernizáció kihívásaira.

Elkészítek továbbá egy kódmodernizációs eszközt is (*transpiler*), amely többek között szemléltetni fogja a módszer működőképességét és hatékonyságát is. Mivel a teljesítmény kritikus kérdés a szoftvermodernizáció területén, ezért a transpiler implementációjának mivolta sem elenyésző része a megoldásnak.

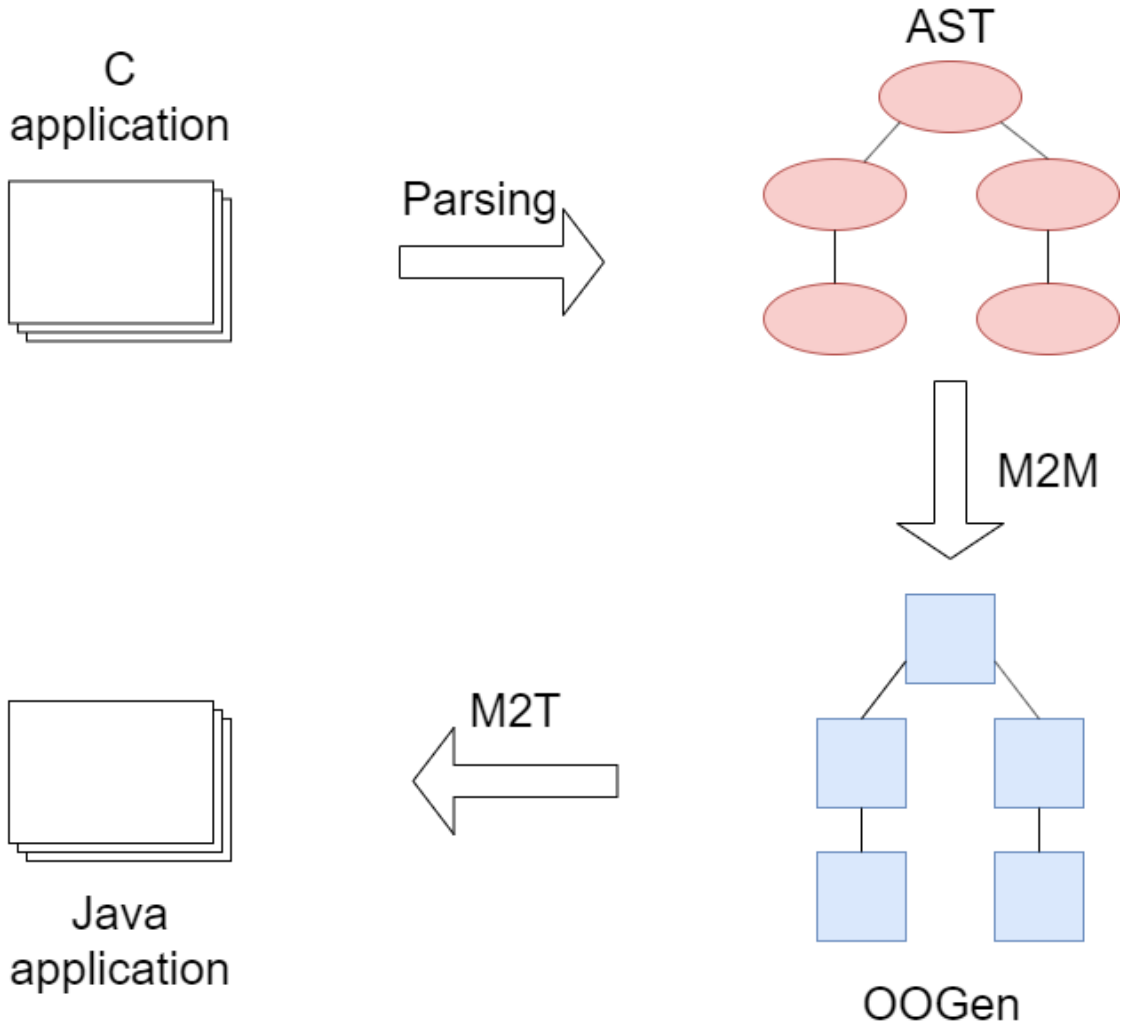
A módszertan objektumorientált szerkezetet kialakító része függetlenül tud működni a forrásnyelvtől és célnyelvtől, azonban természetesen tartalmaz nyelvfüggő részeket is. Mivel a procedurális nyelvek egyik legjelentősebb képviselője a C nyelv, ezért a transpiler forrásnyelve a C lesz. A célnyelvet személyes preferencia alapján Java-nak választom, ez természetesen lehetne tetszőleges más modern nyelv is.

Megjegyzendő, hogy a forrásnyelv és célnyelv konkrét mivolta befolyásolja a transpiler elkészítése során használt technológiákat és környezetet is. Szükség van pl. megfelelő keretrendszerre, amely képes a forrásnyelven írt szoftvereket feldolgozni. Ha ilyen egy adott platformon nem elérhető, akkor a transpilert más környezetben kell elkészíteni.

Természetesen készíthetnénk saját kezűleg is elemzőt (parsert), ez azonban rengeteg felesleges többletmunkával járna.

1.5 Megközelítés

A felvetett problémát dolgozatomban modell-vezérelt eszközökkel vizsgálom. Az adott C nyelvű forráskódot feldolgozom, majd programozásinyelv-független modellt építek belőle. Ez a modell írja le a transzformáció eredményeképp kapott program szerkezeti és egyéb jellemzőit. Ezt bejárva modelltranszformációk sorozatával jutok el a kívánt eredményhez, majd kódgenerálás segítségével előállítom magát a forráskódot az adott célnyelven. Ezt a folyamatot szemlélteti az ***1. ábra***. A dolgozat részletesen bemutatja a transzformáció elvi folyamatát, illetve egy azt megvalósító transpilert is.



1. ábra: A transzformáció folyamata.

Fontos kiemelni, hogy maga a folyamat koncepciója nyelvfüggetlen módon történik. Bármely más forrásnyelv és célnyelv esetén is ugyanezeket a lépéseket végeznénk el.

A transzformáció következő részei függenek a konkrét cél-vagy forrásnyelvtől:

- Elemzés (parsing): Az elemzés folyamata természetesen függ a forrásnyelvtől, hiszen szükség van egy keretrendszerre, amely az adott nyelvű programokat képes feldolgozni.
- Kódgenerálás: A kódgenerálás lépése függ a kijelölt célnyelvtől, mert minden célnyelvhez el kell készíteni a hozzátartozó kódgenerátort is.
- Modelltranszformációk: A modellek kialakítása függhet a forrásnyelvtől és a célnyelvtől is. A modelltranszformációk az objektumorientált nyelvek közös tulajdonságait használják ki, azonban az egyes célnyelveknek lehetnek olyan nyelvi eszközei és sajátosságai, melyeket a többi nyelv nem támogat. Ilyen lehet pl. a C++ operátor túlterhelése. Ugyanez hasonlóan igaz a forrásnyelvekre is a procedurális nyelvek általános közös tulajdonságait illetően.

Megjegyzendő, hogy az objektumorientált nyelvre való áttérés önmagában nem elegendő, fontos a megfelelő objektumorientált szerkezet kialakítása is. Ez alatt azt értjük, hogy hogyan alakítsuk ki az osztályainkat, hogyan azonosítsuk a metódusainkat, illetve hogyan rendeljük azokat osztályokhoz.

Fontos továbbá, hogy a transpiler elkészítése során használt technológiák és környezet is szerves része a megoldásnak. Munkám során az Eclipse *Ecore* [1] alapú metamodellező ökoszisztémáját használom. Ez azért hasznos, mert rengeteg egymással kompatibilis modell-vezérelt eszköz létezik ehhez a környezethez, amelyek hatékonyabbá tehetik a fejlesztés folyamatát. Ilyen eszközök pl. a későbbiekben kifejtett ***Xtext*** és ***Eclipse Modeling Framework (EMF)***.

1.6 A dolgozat felépítése

A ***2***. fejezetben a kidolgozott módszer mögött rejlő elméleti ismereteket, megfontolásokat ismertetem. Bemutatom a forráskód elemzésének és feldolgozásának elméleti folyamatát, illetve röviden felvázolom a transzformáció felépítését.

A ***3***. fejezetben a kódmodernizáció területének korábbi jelentős eredményeiről lesz szó, összehasonlítva a korábbi megoldásokat a sajátommal. Elemzem a kódmodernizáció

kihívásait és bemutatok néhány létező kódelemző eszközt is. A ***4***. fejezet a transpiler elkészítéséhez felhasznált technológiákat ismerteti röviden. Az ***5***. fejezetben mutatom be a felvázolt probléma megközelítésére kidolgozott módszertant. Itt fejtem ki részletesen a transzformáció fő lépéseit.

A ***6***. fejezet az elkészült transpilert mutatja be, ami a módszertan működőképességét és hatékonyságát szemlélteti a gyakorlatban. Kitérek az eszköz felépítésére, képességeire, korlátaira, tesztelésére és a két nyelv közötti konverzió legérdekesebb, nem triviális problémáira.

A ***7.*** fejezetben nyílt-forráskódú szoftverek transzformációjának esettanulmányait mutatom be. Megnézzük, milyen bemenetből milyen kimenetet generál a transpiler.

A ***8***. fejezetben skálázhatósági méréseket végzek a transpiler teljesítményének megállapítására, szintén nyílt-forráskódú szoftvereken. A kapott eredmények alapján elemzem, hogy mitől is függ a transzformátor teljesítménye, illetve becslést adok a teljesítmény tendenciájára is.

Az utolsó, ***9.*** fejezetben összefoglalom a megoldásom lényegét, újszerűségét és értékelem a kapott eredményeket. Végezetül javaslatokat teszek a a megoldás jövőbeni fejlesztésére és bővítésére.

2 Elméleti alapok

2.1 Formális nyelvek

A gyakorlatban programozási nyelvek leírására formális nyelveket [2] használunk, melyeket általában környezetfüggetlen (CF – context free) nyelvtanok segítségével adunk meg. Ezek produkciós szabályok összességével specifikálják az adott nyelven generálható szavak halmazát. A szabályok nemterminális szimbólumokat (változók) és terminális szimbólumokat (szavak) tartalmaznak. A szabályok egymás után való alkalmazásával szavakat generálhatunk. Nyelvünk megengedett kifejezései azok lesznek, amelyeket a szabályok alkalmazásával generálni – szakkifejezéssel levezetni – tudunk.

Az adott programozási nyelv szintaktikáját a formális nyelven megfogalmazott parser szabályok, a lexer működését pedig a lexer szabályok írják le. A szabványos C nyelvhez íródott fordítóprogramok is általában így működnek. A következőkben megnézzük, pontosan mi is a lexer és a parser, és milyen szerepük van a forráskód feldolgozásában.

2.2 A kód feldolgozása

A forráskód elemzése több lépésre bontható, ezek sorrendben: lexikai elemzés, szintaktikai elemzés, szemantikai elemzés [3]. Ebben a fejezetben ezeket a lépéseket nézzük meg részletesebben.

2.2.1 Lexikai elemzés

A forráskód elemzésének első lépése a lexikai elemzés. Ez azt jelenti, hogy a kódot – jellemzően ASCII karakterekből álló szövegként értelmezve – úgynevezett tokenekre bontjuk. Ezek egy-egy logikai egységet jelentenek, amikkel később dolgozni tudunk. A tokeneknek van típusa, illetve rendelhetünk hozzájuk attribútumokat is. Azt a programot, ami a lexikai elemzést elvégzi, lexernek nevezzük. Azt, hogy a lexer milyen szabályok alapján építi fel a tokeneket, általában az adott programozási nyelv fordítóprogramjának lexer szabályai határozzák meg. A lexer szabályokban a tokeneket reguláris kifejezések segítségével adjuk meg.

2.2.2 Szintaktikai elemzés

A szintaktikai elemzés a lexikai elemzés során előállított tokeneket dolgozza fel. Célja a tokenek alapján egy feldolgozható struktúra – általában szintaxisfa – felépítése. Ezen felül a szintaktikai elemzés feladata a szintaktikailag hibás kifejezések felismerése is. Azt, hogy mi számít szintaktikailag helytelennek, általában a parser szabályok specifikálják. Azt a programot, ami a szintaktikai elemzést elvégzi, parser[1]-nek (elemző) hívjuk.

A korábbiakban említésre került, hogy formális nyelveket CF nyelvtannal is leírhatunk, melyek produkciós szabályok összességéből állnak. Minden, a nyelven leírható kifejezéshez levezetésekkel juthatunk el. Egy levezetés a produkciós szabályok egymás után való alkalmazása. Ha ezeket a levezetéseket fa struktúrában ábrázoljuk, akkor egy levezetési fát kapunk. Ez a fa tehát azt tartalmazza, hogyan vezettük le az adott szöveget. Ha ebből a fából elhagyjuk a felesleges, a feldolgozás szempontjából lényegtelen szintaktikai információkat, akkor a kapott struktúrát Absztrakt Szintaxis Fának (Abstract Syntax Tree – AST) nevezzük. A fordítóprogramok jellemzően AST-t generálnak a programok szintaktikai leírására.

2.2.3 Szemantikai elemzés

A szemantikai elemzés célja a forráskód szintaktikailag helyes, de szemantikailag helytelen hibáinak észrevétele. Ilyen lehet például C nyelven egy nem definiált függvény meghívása, vagy egy egész típusú változó sztring értékkel való inicializálása. Mindkét művelet szintaktikailag helyes (a nyelvtan támogatja), de C nyelven nem megengedett.

A szemantikai elemzés az AST-t bejárva validálja a feldolgozott kódot nyelvünk szabályai alapján. Ezután egy modellt készíthet, ami a feldolgozott forráskódot írja le. Ezt azonban nem minden fordítóprogram csinálja így, van olyan is, amely csak AST-t épít. Ezen felül természetesen készíthetők egyéb segédstruktúrák is – pl. szimbólumtáblák –, amikről később részletesebben olvashatunk, jellemzően azonban a feldolgozás célja a programot reprezentáló AST felépítése.

---

[1]A parsing (elemzés) kifejezés sokszor a lexikai, szintaktikai és szemantikai elemzés együttes folyamatára utal.

2.2.4 Példa

Lássuk most egy egyszerű példán keresztül, hogyan működik a lexikai, szintaktikai és szemantikai elemzés. A példa egy rövid forráskódrészlet elemzésén keresztül mutatja be a feldolgozás folyamatát.

Adottak a következő lexer szabályok reguláris kifejezések segítségével:

1. táblázat: A lexer szabályok.

Token	Szabály
T_intLiteral	`(+|-)?[0-9]+`
T_stringLiteral	`’[a-z]+’`
T_if	`if`
T_operatorGreater	`>`
T_operatorEquals	`==`
T_operatorLesser	`<`
T_operatorAssign	`=`
T_openParen	`(`
T_closeParen	`)`
T_openBrace	`{`
T_closeBrace	`}`
T_separator	`;`
T_identifier	`[a-z]+`

A definiált tokenek tehát az egész számok, a sztringek, az *if* kulcsszó, a kisebb, nagyobb és egyenlő operátorok, a nyitó és csukó zárójelek, azok kapcsos változatai, az utasításokat elválasztó pontosvessző, illetve a változó azonosítók. Nyelvünkben az egész számok opcionálisan + vagy – jellel kezdődhetnek, amit korlátlan számú (de legalább

egy) 0-9-es karakter követ. A sztringek aposztróf karakterek között az angol ábécé kisbetűit tartalmazhatják, a változók azonosítói pedig ugyanezeket a karaktereket aposztróf nélkül.

A parszer szabályok EBNF [4] formátumban az alábbiak:

```
statements: (statement)*;
statement: if | instruction;
instruction: variable, T_operatorAssign, operand, T_separator;
variable: T_identifier;
operand: variable | constant;
if: T_if, T_openParen, logical expression, T_closeParen, T_openBrace, body,
     T_closeBrace;
body: (instruction)*;
logical expression: operand, logical operator, operand;
constant: T_intLiteral | T_stringLiteral;
logical operator: T_operatorGreater | T_operatorEquals | T_operatorLesser;
```

Nyelvünk utasítások sorozatát írja le, amelyek elemi utasítások vagy *if* utasítások lehetnek. Az egyetlen megengedett elemi utasítás az értékadás. Ez két oldalból áll egyenlőségjellel elválasztva, a végén pontosvesszővel lezárva. A baloldalon csak változó azonosító szerepelhet, a jobb oldalon pedig operandus, ami vagy változó, vagy konstans lehet. Konstansból kétfélét engedünk meg: egész számot vagy sztringet.

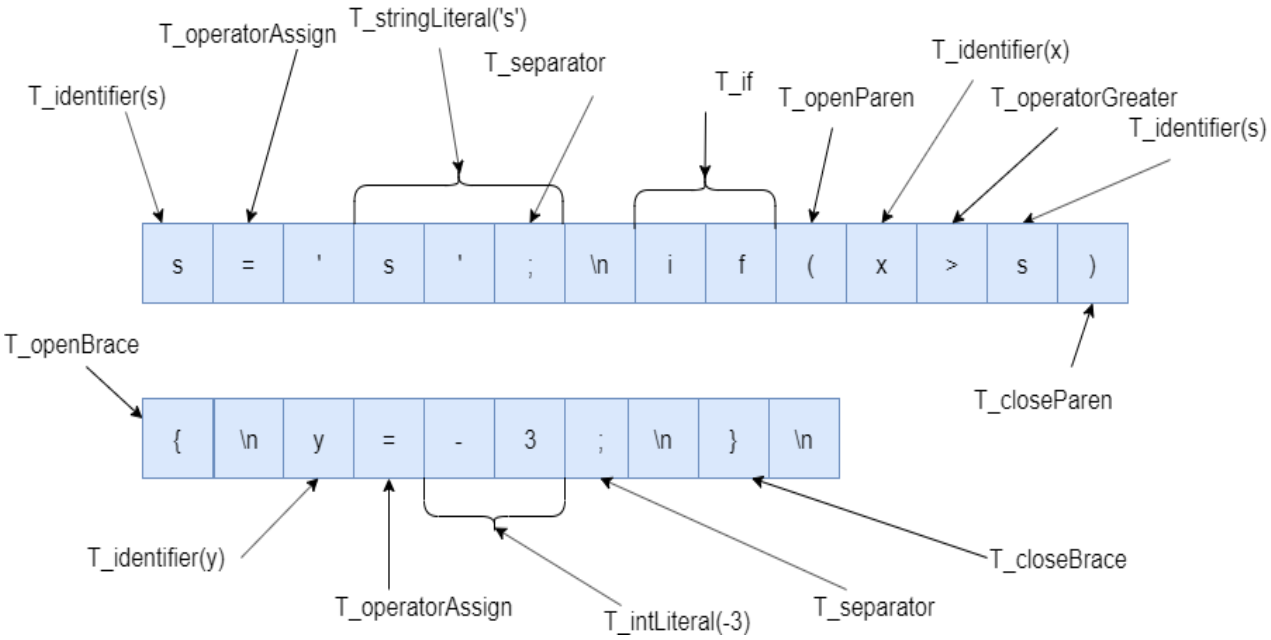
Az *if* utasítás felépítése a következő: az *if* kulcsszót követően egy nyitó és egy csukó zárójel között egy logikai kifejezés szerepel, amely két operandusból és egy logikai operátorból áll. Az operandusok az előbbieknek megfelelően változók vagy konstansok lehetnek, az operátor pedig kisebb vagy nagyobb jel. Ezt követően nyitó és csukó kapcsos zárójelek között az utasítás törzse szerepel, amely tetszőleges számú utasítást tartalmazhat.

Vizsgáljuk az alábbi forráskódot:

```
s = ’s’;
if (x > s) {
     y = -3;
}
```

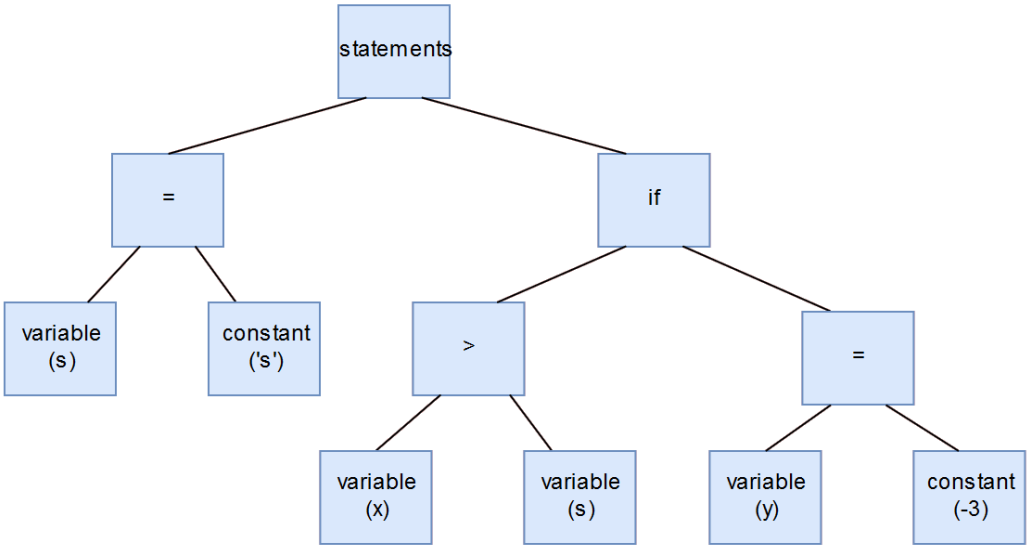
Az első lépés a lexikai elemzés. A forráskódot ASCII karakterekből álló szövegként értelmezve karakterenként vizsgálva feldolgozzuk, és a lexer szabályokra illeszkedő tokeneket keresünk. A whitespace karakterekkel (szóközök és tabulátorok) nem kell foglalkoznunk, mert a folyamatot nem befolyásolják. Az elemzés eredményét a ***2. ábra*** szemlélteti.

A feliratok a tokenek típusát, a zárójelben szereplő értékek pedig azok attribútumait (jelen esetben az értékét) jelentik.



2. ábra: A lexikai elemzés eredménye.

A második lépés a szintaktikai elemzés, amely során a parser szabályok alapján a forráskódot reprezentáló AST-t építjük fel. Először a levezetési fát (parse tree) készítjük el, majd a lényegtelen szintaktikai részletek elhagyásával (pl. zárójelek, pontosvesszők, információt nem hordozó csomópontok) megkapjuk az absztrakt szintaxisfát, amit a ***3. ábra*** szemléltet.



3. ábra: A példa kódhoz tartozó absztrakt szintaxisfa.

Ha a programkódunk szintaktikai hibás lenne, azt a lexikai és szintaktikai elemzés során vennénk észre fel nem ismert tokenek, illetve hibás szerkezetű AST formájában.

Az utolsó lépés a szemantikai elemzés. Ekkor az AST-t bejárva ellenőrizzük, hogy a forráskód betartja-e nyelvünk szemantikai szabályait. Esetünkben például célszerű lehet a nyelvünket úgy értelmezni, hogy logikai operátorokat csak ugyanolyan típusú változókon vagy konstansokon értelmezünk. Ha például x egy korábban létrehozott egész szám típusú változó, akkor a szintaxisfa bejárása során szemantikai hibát találunk, ugyanis az if utasítás logikai feltételében a másik operandus egy sztring.

2.3 Legacy kód transzformációjának folyamata

A folyamat, mely során a procedurális C kódból objektumorientált kódot készítek a következő lépésekből áll. A forráskódot feldolgozom, abból modellt építek, amelyből

kódgenerálás segítségével elkészítem a kódot a megadott célnyelven. A három fő lépés tehát a következő:

1. A kiinduló kódbázist a korábban ismertetett több lépésből álló elemzés (lexikai, szintaktikai, szemantikai) során modellé kell alakítani, amely a program struktúráját, működését leírja.
2. Az így létrejött modell a procedurális kódot írja le, a cél azonban egy olyan modell létrehozása, ami egy objektumorientált programot reprezentál. Ezért az eredeti modell bejárásával létrehozok egy másik, szemantikailag egyező modellt. Ezt a lépést – amikor egy modell transzformációjával egy másikat hozunk létre – a szaknyelv **model-to-model**, röviden **M2M** transzformációnak nevezi. A pontosság kedvéért megemlítendő, hogy nem egy transzformációról, hanem transzformációk sorozatáról van szó. Ezekre azonban tekinthetünk úgy, mintha egy nagy transzformáció ment volna végbe.
3. Ezután már készen áll a végeredményt reprezentáló megfelelő modell. Az utolsó lépés a kódgenerálás, ez hozza létre az alkalmazás tényleges forráskódját az adott célnyelven. Ezt a folyamatot **model-to-text,** röviden **M2T** transzformációnak nevezzük.

3 Kapcsolódó irodalom

Ebben a fejezetben a kódmodernizálás területének kapcsolódó jelentősebb munkáit tekintjük át. Megnézzük, milyen jelentős kihívásokat tartogat a kódmodernizáció szakterülete.

Ismertetem, hogy milyen módszerekkel és milyen megközelítésben vizsgálták már korábban ezt a problémát. Bemutatok továbbá néhány létező kódelemző és kódmodernizáló eszközt is, kitérve erősségeikre és az esetleges hiányosságaikra.

3.1 A kódmodernizáció kihívásai

A kódmodernizálás problémája általános esetben automatizált módon megoldhatatlan feladat, ezért az elkészült megoldások mindig korlátokkal rendelkeznek. A probléma kihívásait [5] foglalja össze alaposan. A szintaxis konverziója egyszerű és könnyen megoldható probléma, azonban a szintaxis konverzióján túl további részproblémákat kell megoldani. Ilyen a paradigmaváltás is, pl. procedurális programkódból objektumorientáltba. Léteznek módszerek ilyen átszervezésre, azonban emberi interakció nélkül nem adnak kielégítő eredményt.

A másik probléma a különböző nyelvek eltérő elemkészlete. A forrásnyelv bizonyos elemei nem képezhetők le a célnyelv natív elemeire, ezért ezeket az átalakítás során szimulálni szükséges. Pl. a C++ nyelv lehetőséget ad operator overloadingra, de a Java nem. A konverzió nemcsak a paradigma szintjén fontos, hanem az alacsonyabb szintű nyelvspecifikus idiómák szintjén is. Egy jól működő megoldástól a felhasználói azt várják el, hogy olyan minőségű kódot adjon, mintha azt eleve a célnyelven írták volna. Az adatok konverziója sem triviális feladat. Például a C++ nyelvben léteznek pointerek, nincs boolean típus, és a különböző típusok mérete függ a használt platformtól. Ezzel szemben a Java nyelvben nincsenek pointerek, van boolean típus, és a típusok méretét szabvány írja elő.

Egyes esetekben a megoldást csak az adattípusok emulációja jelentheti, ha ragaszkodunk a funkcionálisan teljesen ekvivalens megoldáshoz. Ez viszont műveletek emulációját is megkívánná, illetve ellent mondana annak az elvárásnak, hogy a kimenet olyan minőségű legyen, mintha a célnyelven írták volna. Egy további kihívás, hogy némely nyelveknek

több dialektusa létezik, és ezek között a szintaktikai különbségek gyakran kicsik, de a szemantikai különbségek annál jelentősebbek.

A kihívásokat összefoglalva azt mondhatjuk, hogy reálisan csak interaktív vagy utólagos refaktorálást igénylő megoldást tűzhetünk ki célul. A dolgozatban az utóbbit valósítjuk meg, és különös fókuszt helyezünk arra, hogy a kimenet a célnyelv elemeit minél természetesebb módon használja. Ha a konverzió által nem kielégíthetően kezelt részek a kimenetben jól érthetőek és könnyen refaktorálhatóak maradnak, akkor ennek a részleges megoldásnak a gyakorlati haszna is jelentősen magas lehet.

[6] továbbá azt is megállapítja, hogy a parserek fejlesztéséhez használt technikák nem modulárisak és kompozicionálisak, ezért a modernizációs kísérletekben nehézkesen használhatók. A dolgozatban bemutatott módszerben ez a probléma nem állt fent, hiszen a módszer nem a szintaxis puszta átalakításán alapul, hanem először egy magas szintű szemantikus modellt épít a CDT parserének újrafelhasználásával.

3.2 Megközelítések és módszerek

Az idézett megoldások két csoportra oszthatók aszerint, hogy milyen alapvető megközelítésben vizsgálták a problémát. Az egyik csoport tagjai a *cluster analysis* [7] módszerével szisztematikusan folyamatosan szűkölő csoportokra bontják a szoftver entitásait (adatstruktúrák, függvények, globális változók) a köztük lévő függőségek alapján. Céljuk, hogy a kialakult csoportok között minimálisak legyenek a függőségi viszonyok. Az így kialakított csoportok alkotják az alkalmazás osztályait.

Bár mindegyik ide tartozó munka hasonlóan közelíti meg a problémát, vannak jelentősebb eltérések is köztük. [8] végeredménye nem kód, hanem modell, amelyet a klaszterek egyesítésével alakít ki. [9] egy saját heurisztikus algoritmust dolgozott ki a klaszterek kialakítására, amely nagy méretű szoftvereken hatékonyabb lehet, mint a másik két megközelítés. [10] a *cluster analysis* mellett a *concept analysis* [11] szakterületét is felhasználja munkájában.

A másik csoport tagjai ezzel szemben többnyire nem veszik figyelembe a kódrészletek viszonyait, a csatolás és a kohézió kérdéseit. [12] egy szisztematikus megközelítést ad PL/I nyelvről Javára történő konverzióra egy esettanulmányon keresztül. [13] és [14] megoldásában közös, hogy mindkettő magasabb szintű segédstruktúrákat generál a

transzformáció támogatásához. [13] hibás tervezés nyomait keresi a kódban, [14] az alkalmazás követelmény analízisét használja fel megoldásában.

3.2.1 Klaszterező megoldások

A [8] által bemutatott megközelítés első lépésként egy olyan XML állományt épít az AST-ből, amely egy nyelv-független, procedurális alkalmazás szintaktikai sajátosságait írja le. A második lépés egy inkrementális algoritmus használata, ami a programot több, kisebb méretű összetevőre (cluster) bontja. Ezeket vizsgálva több kisebb, úgynevezett Domain Object Modelt (DOM) épít. Végül ezeket „összerakva" alakul ki az alkalmazás végső modellje. A megoldás végterméke tehát egy modell, ami a modernizált alkalmazást írja le. A dolgozatban bemutatott megoldás szintén modellt készít, viszont a kódgenerálás lépését is elvégzi. A végterméke így tényleges kód, nemcsak egy modell.

[9] a megoldás első lépéseként az AST-t bejárva kigyűjt minden olyan információt, amire szüksége van. Ezekből entitásokat épít, amelyeken a második lépésben dolgozni tud. Ezután következik az osztályok kialakítása. Erre a célra a szerzők egy heurisztikus partícionáló algoritmust dolgoztak ki, amely az entitások között úgynevezett távolságot (distance) definiál. Az algoritmus futása során az entitások vizsgálatán keresztül ciklikusan korrigálja a távolság értékeket, és azokból az entitásokból, amelyek távolsága egymáshoz képest nagyon kicsi, clustereket épít. Az algoritmus tehát az összes entitást tartalmazó kiinduló halmazt entitásokat tartalmazó részhalmazokra (clusterre) bontja. Ezek a részhalmazok adják az objektumorientált alkalmazás szerkezetét.

[10] a cluster analysis és concept analysis szakterületét hívja segítségül az objektumok azonosításához. Ez a megközelítés hasonlít a dolgozatban bemutatott módszerre abban, hogy a funkcionalitás tekintetében összefüggő részeket próbálja azonosítani, és ezek alapján alakítja ki az osztályokat. A dolgozatban bemutatott megoldás nemcsak az objektumok azonosítására fókuszál, hanem a feladat további részproblémáit is tárgyalja, valamint az objektumok azonosításához bemutatott megoldás is bővebb, hiszen a struktúrákat és függvényparamétereket is felhasználjuk az objektumok azonosítása során, nemcsak a hívási relációkat.

3.2.2 Egyéb megközelítések

[12] egy PL/I nyelvről Javára történő konverziót ír le esettanulmányként. A konverzió jól beazonosítható lépésekben, egy szisztematikus keretben történt. Ez a módszer más konverziókhoz is jó kiinduló alapot adhat, de a konverzió itt sem teljes. Az objektumok létrehozásánál az adatokra támaszkodik, és a kódrészletek viszonyait, a csatolás és a kohézió kérdéseit nem veszi figyelembe. A kimeneti kód formailag ugyan objektumorientált, de a tényleges paradigmaváltás nem történt meg kielégítően. Ezen kívül a kivételkezelést is utólag kell megoldani a programozónak a konverzió után.

[13] olyan keretrendszert készít, amely félautomata módon képes elvégezni a transzformációt. Az itt bemutatott módszer elsősorban hibás tervezés nyomait keresi (pl. redundáns vagy duplikált megoldások), és ezeket kijavítva szervezi osztályokba és metódusokba az eredeti alkalmazást. Ehhez különböző adatstruktúrákat – pl. adatfolyam-gráfot vagy állapotgépet – épít a programból, és ezek segítségével vizsgálja a kódot.

[14] a transzformáció során a minél magasabb szintű újrahasználhatóság elérésére törekszik. Ehhez feltételezi, hogy az alkalmazás követelmény elemzése (requirement analysis) rendelkezésre áll, ugyanis ezt az információt felhasználja a transzformáció során. A folyamat különböző magasabb absztrakciós szintű elemeket készít a transzformáció segítésére (pl. ER[2] diagramm, dataflow[3] diagram), majd könnyen újra hasznosítható komponensekre próbálja bontani a programot, az objektum-orientált paradigmákat szem előtt tartva.

3.3 Kódelemző eszközök

3.3.1 mtSystems

[15] egy svéd szoftverfejlesztő cég, amely C-ről Java nyelvre fordító kódmodernizációval foglalkozik. Bár az eszközük nem nyílt forráskódú, rendelkezésre bocsájtanak egy webes felületet, ahol egyszerű kódrészleteken kipróbálható a működése. Elérhető továbbá a *Vim*

[2] https://www.smartdraw.com/entity-relationship-diagram/

[3] https://www.cs.uct.ac.za/mit_notes/software/pdfs/Chp06.pdf

szövegszerkesztő transzformáltjának forráskódja, amely az eszköz képességeit demonstrálja.

A készítők célközönségüknek nemcsak a legacy szoftvereket tekintik. Ide sorolják a következő eseteket is:

- Ha Java nyelven dolgozunk és egy szükséges könyvtár csak C nyelven elérhető, akkor azt transzformálhatjuk Java-ra és már használhatjuk is.
- Ha C könyvtárakat fejlesztünk, akkor azokat karbantartási többletmunka nélkül Java-kompatibilissé tehetjük.

Ez azért fontos, mert rávilágít, milyen hasznos tud lenni akár más területeken is egy ilyen szoftver.

Az eszköz jól kezeli a C szintaktikai elemeit, azokat szinte hibamentesen képes Java kódra transzformálni. Bizonyos elemeket nem sztenderd Java-beli nyelvi elemekre képez le, hanem definiál a problémához segédosztályokat. Ilyenek pl. a különböző konténerosztályok, amelyeket bizonyos típusok leképzésénél használ.. Képes kezelni a makrókat is és még *goto* utasításokat is támogat.

A program bár képes automatikusan, szintaktikailag helyes Java kódot generálni, mégis van egy jelentős hiányossága. Az objektumorientált szerkezet kialakításával nem foglalkozik, így paradigmaváltást egyáltalán nem végez. Ez az előzőekben felsorolt másik két esetben nem számít, viszont kódmodernizálásnál elemi szintű hiányosság.

3.3.2 Tangible Software Solutions

[16] kifejezetten transpilerek gyártásával foglalkozik. Összesen hatféle konverter közül választhatunk, amelyek VB.NET, C#, C++ és Java nyelvek között konvertálnak. Mindegyik eszköznek egy limitált (maximum 100 sor / file) verziója elérhető ingyenesen.

Bár ezek a szoftverek ránézésre hasonlóak a kódmodernizáló eszközökhöz, itt inkább a szintaktikai konverzió a lényeg. Ezek a nyelvek ugyanis alkalmasak modern környezetben futó szoftverek üzemeltetésére. Ezért itt nem paradigmaváltás a cél, hanem az adott nyelvek közötti automatikus átjárás biztosítása.

Példaképpen tekintsük a C++-ról Java-ra fordító eszközt. Az eszköz félautomata módon működik. Bár a szintaktikával jól boldogul, rendelkezik bizonyos limitációkkal. Ezeknek

a nagyrésze a korábban említett nyelvi különbségekből fakad. Ezek a teljesség igénye nélkül a következők:

- A beépített C/C++ függvényhívások nagyrészét nem kezeli a program.
- A pointeraritmetikával nem tud mit kezdeni, mert Java-ban nincs ehhez hasonló nyelvi elem. Ez a dolgozatban bemutatott megoldás esetén is így van.
- A feltételes fordításokat (Preprocessor utasítások) refaktorálni kell ekvivalens működésű kódra transzformálás előtt, mert a Java-nak nincs preprocesszora.
- Beágyazott assembly kódok, többszörös öröklés, *friend* függvények és osztályok szintén nem képezhetők le, mert Java-ban ezek nem megengedettek.

Ezeken a példákon jól látszik, hogy a nyelvi különbségek komoly korlátokat tudnak szabni a transzformáció automatizálásának lehetőségeire.

3.3.3 Ericsson Codecompass

[17] egy nyílt forráskódú *code comprehension*[4] eszköz, amely C és C++ kódot képes elemezni. Az eszköz képességeiről [18] ad összefoglalót.

Az eszköz az előző fejezetben bemutatottakat elvégezve elemzi a kódot és az így kinyert információt egy adatbázisban tárolja. Tulajdonképpen egy statikus kódanalizáló eszközről van szó: a forráskód elemzésén keresztül rendkívül sokféle információhoz juthatunk a kóddal kapcsolatban. A *Codecompass* többek között generál vezérlésfolyam-gráfot a függvényhívásokból, illetve különböző kódmetrikák segítségével méri a forráskód minőségét.

Az eszköz a kód vizsgálatán kívül a kód verziókezeléséhez használt *git historyt* is eléri egy git parser segítségével. Ezt felhasználva átlátható módon összegzi az adott fájlok szerkesztési előzményeit, a szerzők változtatásait. Ezen felül úgynevezett *browsing historyt* tesz elérhetővé, amely a kódban való navigációt teszi könnyebbé. Ez a kód emberi megértését könnyíti meg.

Az eszköz nagy előnye, hogy viszonylag könnyen integrálható más eszközökhöz. [18] ennek szemléltetésére bemutatja a *Codecompass* integrációját a szintén *Ericsson* által

[4] https://lukasatkinson.de/2017/what-is-a-source-code-comprehension-tool/

kifejlesztett *CodeChecker* eszközzel [19]. Az integráció eredményeként pl. amikor a *CodeChecker* egy potenciális hibát talál a programban, akkor a *Codecompass-en* megtekinthetjük az adott hibát előidéző vezérlési utat is.

Ezek alapján érdekes jövőbeni iránya lehet a transpilernek a *Codecompassal* (vagy más hasonló kódelemző eszközzel) való integráció. Egy kódelemző eszköz által összegyűjtött információk hasznosak lehetnek a transzformáció során.

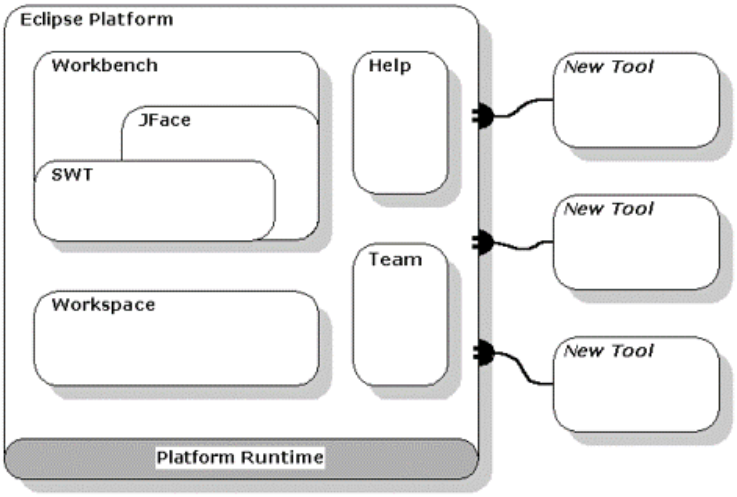
4 Kapcsolódó technológiák

Ebben a fejezetben a megoldás során felhasznált technológiákat tekintjük át röviden.

4.1 Eclipse és OSGi

Az Eclipse[5] [20] kezdetben egy integrált fejlesztőkörnyezet (IDE – Integrated Development Environment) volt. Az idő során azonban fejlődésen ment keresztül, sok komponense hasznosnak bizonyult más jellegű alkalmazások fejlesztéséhez is. Így alkalmazásplatform lett belőle, amely pluginek használatát támogatja. Ez azt jelenti, hogy az adott szoftver az Eclipse-en belül tud futni, kihasználva a platform nyújtotta szolgáltatásokat. Az Eclipse így rendkívül rugalmas és testreszabható, különböző disztribúciói eltérő plugineket tartalmazhatnak. Ráadásul saját pluginek is készíthetők és használhatók. A procedurális kód transzformációjának gyakorlati prototípusa is Eclipse pluginként működik.

Az Eclipse felépítését a ***4. ábra*** szemlélteti.



4. ábra: Az eclipse moduláris felépítése.

[5] https://www.eclipse.org/

Az Eclipse az OSGi-re [21] épül, amely egy nyílt szabvány moduláris rendszerek integrációjához. A pluginek bővíthetik más pluginek funkcióit, illetve függhetnek is egymástól. Az OSGi szabvány ezeket a függőségeket egy manifeszt fájlban (MANIFEST.MF) definiálja. A pluginek kiterjesztési pontokat is megadhatnak, amelyek funkciókat szolgáltatnak más pluginek számára. Ezeket a plugin.xml állományban kell megadni.

4.2 C Development Tooling (CDT)

A CDT [22] az Eclipse platformhoz szolgáltat funkcionalitásokat (plugineket), amelyek C és C++ projektek fejlesztéséhez biztosítanak integrált fejlesztőkörnyezetet. Habár önmagában nem tartalmaz C/C++ fordítót, lehetőséget biztosít azok Eclipse környezetbe való egyszerű integrációjához. A CDT a megszokott funkciókat biztosítja Eclipse környezetben C és C++ fejlesztés számára: pl. projektek létrehozása, kezelése, keresés a forráskódban, content assist támogatás. Nagy előnye, hogy Eclipse plugineket használhatunk C/C++ fejlesztés közben.

A CDT a korábbiakban bemutatott folyamatnak megfelelően fel tudja dolgozni az adott (C vagy C++) projektet, majd abból AST-ket tud építeni, melyek leírják a projekt szerkezetét és jellemzőit. Ez azért előnyös, mert ezeket egy API-n keresztül el tudjuk érni programozottan. Így futásidőben felhasználhatjuk és manipulálhatjuk a projektünk szinte minden tulajdonságát. Az AST-k bejárásán keresztül elérhetjük a forrásfájlokat, az azokban definiált függvényeket, változókat, utasításokat.

4.3 Java Development Tools (JDT)

A JDT [23] a CDT-vel analóg módon plugineket szolgáltat Eclipse platformra, melyek egy Java projektek fejlesztésére alkalmas integrált fejlesztőkörnyezetet valósítanak meg.

A készítők a JDT pluginjeit a következő kategóriákba sorolják:

1. A JDT APT pluginek annotációk feldolgozásához szükséges funkciókat biztosítanak Java 5-ös vagy későbbi verziójú projektek számára.
2. A JDT Debug a Java projektek hibakeresés funkcióját valósítja meg.
3. A JDT UI különböző refaktorálást elősegítő és áttekintő funkciókat biztosít grafikus felületek segítségével. Ilyen például maga a Package Explorer, a típus

hierarchia megtekintése, vagy elemek átnevezése (rename) az egész projektben található összes hivatkozás lecserélésével együtt.

4. A JDT Text pluginek a Java forráskód szerkesztőjét valósítják meg.
5. A JDT Core biztosítja a Java fordítót az Eclipse platformra. Ezen felül, ahogy a CDT C és C++ projektekre, úgy a JDT Core is szolgáltat API-t Java projektek programozottan történő elemzésére, manipulálására, létrehozására. Ennek segítségével egy Eclipse Java projektbe szervezhető a kódgenerátor által előállított Java kód.

4.4 Eclipse Modeling Framework (EMF)

Az EMF[6] [24] az Eclipse metamodellező keretrendszere. A metamodell a modell modelljét jelenti, amelynek egy konkrét példánya pedig egy modell lesz. A kód transzformációja során építendő programozásinyelv-független objektumorientált modell egy EMF segítségével készített metamodell példánya lesz.

Az EMF segítségével tehát metamodelleket tudunk létre hozni. Ha ezzel megvagyunk, akkor abból az EMF közreműködésével kódot kell generálnunk, hogy használni tudjuk azt. A kódgenerálás paraméterezhető is, megadható pl. a generált kód Java-verziója. Ezek a paraméterek a metamodelltől függetlenül, egy másik modellben, a „genmodel” kiterjesztésű generátormodellben tárolódnak.

Ha elkészültünk a generálás paraméterezésével is, akkor generálhatók a metamodellünk osztályaihoz tartozó modellosztályok. Minden osztályhoz tartozik egy interfész, és az azt implementáló Java osztály. Ezeknek az osztályoknak a példányai lesznek a metamodellünk példányai, tehát egy példány egy konkrét modellt fog jelenteni.

Az EMF modell használatának legnagyobb előnye a vizsgált problémakör szempontjából, hogy számos kiforrott, Eclipse környezetben pluginként futó modell-vezérelt eszköz létezik, amely támogatja az EMF modelleken való munkavégzést. Erre jó példa a következő alfejezetben bemutatott eszköz.

[6] https://www.eclipse.org/modeling/emf/

4.5 Xtext

Az Xtext[7] [25] egy olyan keretrendszer, melynek segítségével szakterületi nyelveket hozhatunk létre. A szakterületi nyelv (Domain Specific Language – DSL) [26] egy adott problémakör megoldására specializált nyelv.

Az Xtext egy, az ANTLR-en [27] alapuló parser generátort valósít meg. A parser generátorok egy nyelvtant várnak, amelyek megadják a lexer által generálandó tokenek leírását reguláris kifejezésekkel, illetve a nyelvünk szintaktikai szabályait. Ez alapján a generátor elő tudja állítani az adott nyelv lexerét és parserét. Ezt a nyelvtant, amely leírja a nyelvünket, metanyelvnek nevezzük. Tulajdonképpen a metanyelv is szakterületi nyelvnek tekinthető, amellyel nyelveket lehet leírni.

Az Xtext nem csupán egy egyszerű parser generátor, azon kívül többletfunkciókat is biztosít. Egyrészt, az Xtext nem szintaxisfát készít, hanem egyből egy EMF metamodell példányát adja vissza. Másrészt az elkészülő parsert Eclipse pluginokként generálja, és magán az elemzőn kívül mást is készít. Ilyen például a kódgenerátor, a formázó stb.

Az Xtext-et munkám során a forrás-és célnyelvek közötti alapvető API konverziók leírására használom. Xtext segítségével kifejleszthető egy szakterületi nyelv, amellyel az alapvető műveletek közötti konverzió leírható. Ebből a nyelvből programozottan modell generálható, ami alapján ezeket a hívásokat át lehet transzformálni a célnyelvnek megfelelő hívásokra. Mindezt részletesen az ***5.3.*** fejezetben ismertetem.

[7] https://www.eclipse.org/Xtext/

5 A kidolgozott módszer bemutatása

Ebben a fejezetben a procedurális kód objektumorientált kódra való transzformálásának általam kidolgozott megközelítését mutatom be. Röviden kitérek a forráskód elemzésére és feldolgozására, majd részletesen ismertetem az objektumorientált modell felépítésének lépéseit. A folyamat utolsó lépése a kódgenerálás, ami az alkalmazás tényleges kódját hozza létre.

5.1 C projekt feldolgozása

A transzformáció bemenete egy C projekt, ami tetszőleges számú forrásfájlból és fejlécfájlból állhat. Az első lépés a kód feldolgozása, az azt reprezentáló AST felépítése. Ezt a korábbiakban ismertetett elemzés (parsing) folyamata végzi el.

Az absztrakt szintaxisfa felépítése után annak bejárásával nyerhető ki a C kódról minden olyan információ, ami a modell felépítéséhez szükséges. Ezek alatt a projekt valamennyi forrásállományának tartalmát értjük, mint például a függvénydefiníciók, azok törzsei, a létrehozott struktúrák és tagváltozóik, a globális változók, stb. A továbbiakban a függvényekre, globális változókra és struktúrákra *entitásként* hivatkozok.

5.2 A modell felépítése

Ahhoz, hogy a procedurális kódbázist jól karbantartható, áttekinthető és könnyen bővíthető kóddá alakítsuk, nem elég ugyanazt a szerkezetet egy objektumorientált nyelven leírni. Ennek érdekében meg kell adni, hogyan épüljön fel a modernizált alkalmazás szerkezete, vagyis hogy az eredeti kódot milyen elvek mentén és milyen módszerek alkalmazásával szervezzük osztályokba.

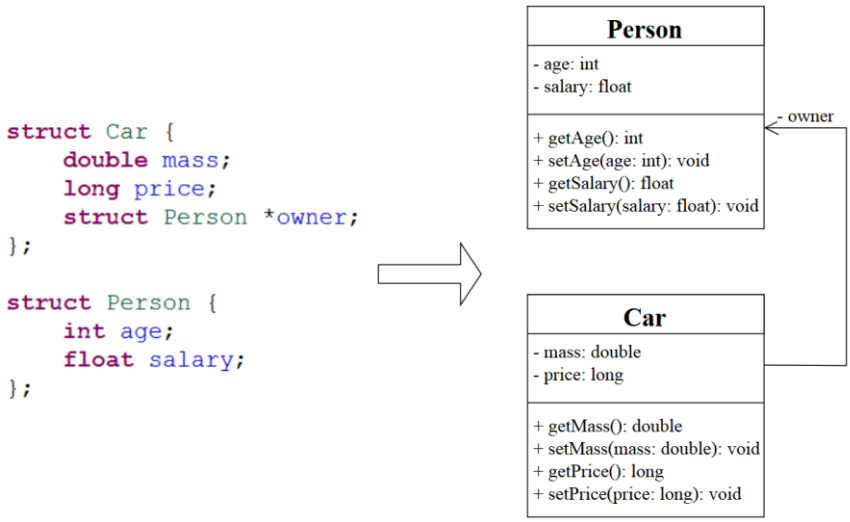
A továbbiakban bemutatok egy algoritmust lépésenként, amely az osztályhierarchia kialakítását végzi el különböző szempontok vizsgálata alapján. Az algoritmus több lépésből álló vizsgálatot végez, és ha egy vizsgálat végén egy entitást már osztályhoz rendelt, akkor az átkerül a befejezett entitások halmazába. Így a következő vizsgálatot már csak azokon az entitásokon végzi el, amelyek még nem kerültek osztályba. A vizsgálatok tehát egy folyamatosan szűkülő adathalmazon dolgoznak. Fontos megemlíteni, hogy előfordulhat, hogy egy entitást több elemzés is besorolhatna, akár

különböző osztályokba is. Ezért a vizsgálatokat eredményük prioritásának csökkenő sorrendjében végzem.

Minden olyan entitást, amit nem sikerül a vizsgálatok során osztályhoz rendelni, egy globális osztályba szervezek. Ide kerül természetesen az alkalmazás belépési pontja is.

5.2.1 Struktúrák leképzése

Az osztályok kialakításának első lépése, hogy minden, a C kódban definiált struktúrát egy osztályra képzünk le az objektumorientált kódban. Az osztály tartalmazza a struktúra összes tagját privát tagváltozóként, amelyek értékét az adott osztályban felvett publikus getter és setter metódusokon keresztül érhetjük el, illetve állíthatjuk más értékre. A struktúrák osztályba való leképzését az ***5. ábra*** szemlélteti egy példán keresztül.



5. ábra: Struktúrák leképzése osztályokká.

A példán látható struktúrák tagjainak típusai tájékoztató jellegűek, csak a leképzés szemléltetését szolgálják. A C típusainak kezeléséről később, a ***6.4.1***. fejezetben írok.

Természetesen a publikus getter és setter metódusok generálása önmagában nem szavatolja a tökéletes egységbezárást és konzisztenciát az adatok között. Ehhez mélyebben kellene ismerni az alkalmazás szemantikáját és követelményeit. Amennyiben ilyen információ a kódban nem áll rendelkezésre, ezt automatizálni nem lehet.

5.2.2 Paraméterlista és visszatérési érték vizsgálata

Az első lépésben az entitások közül a struktúrákból készítünk osztályokat, a függvényekkel azonban még nem foglalkoztunk. A második lépés célja ennek megfelelően a függvények hozzárendelése az előző lépésben kialakított osztályokhoz.

Ennek érdekében a függvények paramétereit és visszatérési értékeit vizsgálhatjuk struktúra típusok szempontjából.

Egy függvény struktúra típusú paramétere azt jelzi, hogy a függvény valamire használja az adott típust. Mivel a függvény valamilyen szempontból felhasználja ezt a típust, érdemes lehet metódusként az ehhez a típushoz tartozó osztályba szervezni, hiszen az adott típussal dolgozó művelet logikailag összetartozhat az adatokkal.

Ha tehát egy függvény paraméterlistája struktúra típusú változót vár, akkor ez a függvény potenciálisan hozzáadható az adott struktúrához tartozó osztályhoz. Ezt viszont csak akkor lehet biztosan kijelenteni pusztán paraméterlista vizsgálattal, ha csak egyfajta ilyen változó szerepel a listán. Ha több is szerepelne, akkor nem lenne egyértelmű, hogy melyikbe célszerűbb szervezni a függvényt[8].

A struktúra típusú visszatérési érték arra utal, hogy a vizsgált függvény az adott típusból vagy frissíti egy változó értékét, vagy egy újat hoz létre és ad vissza. Mindkét esetben célszerű az ehhez a típushoz tartozó osztályba szervezni metódusként a vizsgált függvényt, ugyanis az objektumorientált szemlélet szerint egy típus adatai és a típus állapotának megváltoztatásáért felelős műveletek egybetartoznak. Ezt nevezzük egységbezárásnak. A visszatérési érték ezért alapvetően erősebb jelnek tekinthető, mint a paraméter. Ha mindkét helyen van struktúra típus, akkor a visszatérési érték osztályába szervezendő a függvényt.

Az előbbieknek megfelelően azt, hogy a visszatérési érték és paraméterlista vizsgálat eredményeinek különböző kombinációi esetén tud-e következtetni az algoritmus, és ha igen, akkor hogyan dönt, a ***2. táblázat*** foglalja össze.

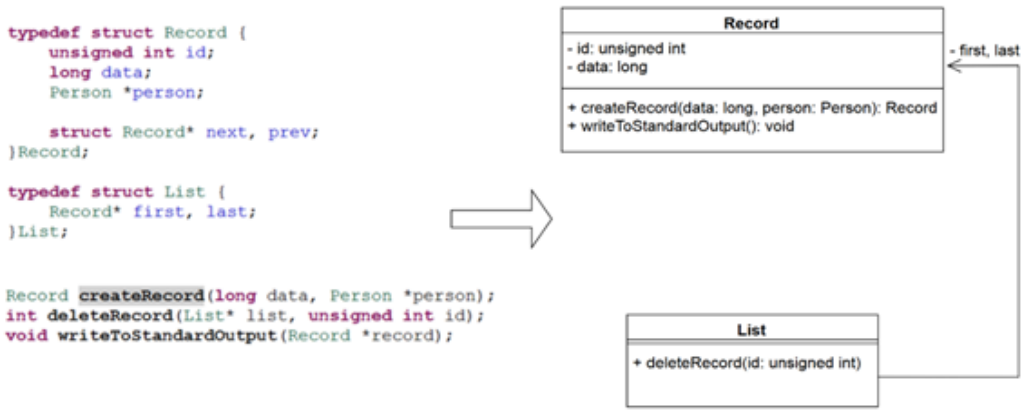
---

[8] Létezik olyan programozási konvenció, hogy a paraméterek „fontosságuk” sorrendjében vannak felsorolva. Ez alapján ebben az esetben is lehetne dönteni, viszont ez egy erős feltételezés lenne, amit közel sem biztos, hogy betart a vizsgált alkalmazás.

2. táblázat: A paraméterlista és visszatérési érték vizsgálatának esetei.

	Paraméterlistában nincs struktúra típus	Paraméterlistában egy struktúra típus van	Paraméterlistában több struktúra típus van
Visszatérési érték nem struktúra típus	Nem célszerű osztályba szervezni	Az adott struktúra típusba célszerű szervezni	Nem tud dönteni
Visszatérési érték struktúra típus	A visszatérési érték típusába célszerű szervezni	A visszatérési érték típusába célszerű szervezni	A visszatérési érték típusába célszerű szervezni

Azokban az esetekben, amelyeknél sikerül azonosítani a célosztályt, az algoritmus el is végzi a metódus hozzáadását az osztályhoz. Ahol nem tud dönteni, vagy arra a következtetésre jut, hogy egyik meglévő osztályba se célszerű beszervezni az adott függvényt, ott nem csinál semmit, tehát a függvény további vizsgálat bemenete lehet.



6. ábra: Paraméterek és visszatérési érték vizsgálata.

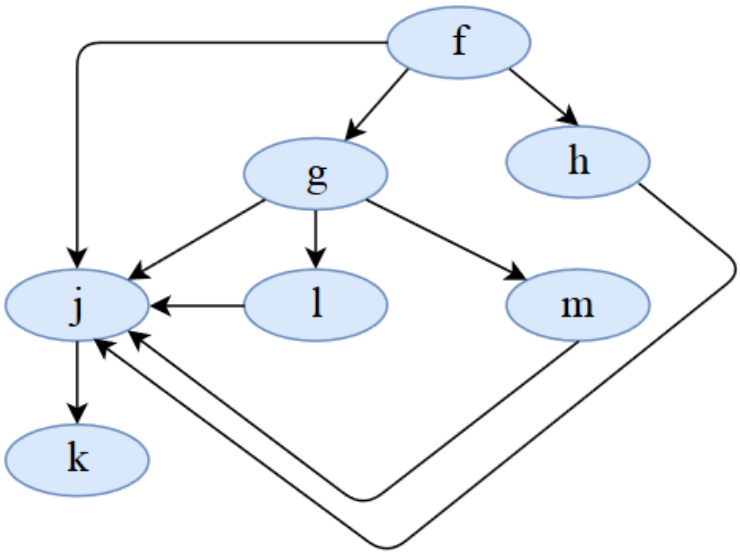
A ***6. ábra*** egy egyszerű példán keresztül mutatja be a paraméterek és visszatérési érték vizsgálat mögötti ötletet. Adott egy láncolt lista és néhány, a rekordelemeken, illetve a listán műveletet végző függvény. Az első függvény egy konstruktorfüggvény lehet, a második töröl egy elemet a listából, a harmadik pedig kiír egy listaelemet a sztenderd kimenetre. Az első a visszatérési érték miatt a Record osztályba szerveződik, a második és harmadik pedig a List és Record paramétertípusok miatt az adott osztályba kerülnek.

5.2.3 Hívási láncok keresése

A harmadik lépés bemenete minden olyan függvény, amely az előző lépésben nem került osztályba. A vizsgálat célja további, bizonyos szempontból összetartozó függvények osztályba szervezése.

Ennek érdekében egy hívási gráfot (callgraph– CG) építek, amely a program vezérlésének „útját” jelöli. Ez egy irányított gráf, amelynek csomópontjai a bemeneti függvények, egy él pedig valamely ***f*** csomópontból ***g*** csomópontba mutat pontosan akkor, ha ***f*** futása során legalább egyszer meghívja ***g***-t.

A ***7. ábra*** egy ilyen CG-t szemléltet.



7. ábra: Egy példa CG.

A CG tehát minden bemeneti függvényre tartalmazza, hogy az mely függvényeket hívja meg futása során. A vizsgálat során hívási láncokat keresek, és bizonyos szempontok alapján a gráf egy-egy összefüggő részgráfjából (hívási láncból) készítek egy-egy osztályt.

Jelölje egy ***f*** csomópontra valamely más ***g*** csomópontokból ***f*** felé mutató élek számát f_{in}. Ezzel a terminológiával élve az algoritmus olyan csomópontokat keres, amelyekre f_{in} alacsony. Ez lesz az algoritmus kiinduló (gyökér) csomópontja. Ha az algoritmus talál egy ilyen ***f*** függvényt, akkor ***f*** gyerekeit is megvizsgálja bemenő fokszám szempontjából, majd azok gyerekeit, stb. Egy csomópont gyerekeit akkor nem vizsgálja tovább, ha nincs

neki, vagy ha az adott csomópont f_{in} értéke túl magas. Ha a hívási láncban található függvények száma meghalad egy minimális értéket, akkor ezek a függvények kerülnek egy osztályba. Ezt a minimális méretet jelölje s_{min}.

Az algoritmus tehát arra törekszik, hogy olyan leghosszabb hívási láncokat keressen, amelyek mindegyik tagját viszonylag kevés függvény hívja az alkalmazásban. Ez azért eredményez előnyös osztályszerkezetet, mert egyrészt a hívási lánc miatt ezek a függvények logikailag egybetartozhatnak, másrészt az alacsony f_{in} értékek biztosítják az osztály laza csatolását az alkalmazás többi része felé, így azok kevésbé fognak függeni az osztálytól. A minimális osztályméret biztosítja, hogy ne alakuljon ki sok kicsi, aránylag kevés függvényt tartalmazó osztály.

Azt kell még megadnunk, hogy mi számít elég alacsony határértéknek f_{in} esetén, illetve, hogy mennyi legyen a minimális osztályméret. Ezeket az értékeket a programban található összes függvény számától tehetjük függővé. Jelölje ezt a számot **N.** Ekkor az f_{in} értékek maximálisan megengedett értéke $maxf_{in} = \lceil 0{,}2 * \mathbf{N} \rceil$. Egy f_{in} tehát határértéken belül van, ha $f_{in} \leq maxf_{in}$, szemléletesen ha nem haladja meg a függvények számának 20%-át (felfele kerekítve). Hasonlóan, a minimális osztályméret $s_{min} = \lceil 0{,}1 * \mathbf{N} \rceil + 1$. A +1 azt a célt szolgálja, hogy nagyon kevés hívással rendelkező alkalmazások esetén legalább két metódus kerüljön egy osztályba.

Az algoritmusom pontos működését a következő pszeudokód szemlélteti, ahol V(CG) a csúcsok halmazát, E(CG) az élek halmazát, $|H|$ egy H halmaz elemszámát, {} pedig az üres halmazt jelenti.

***Algoritmus** hívási láncok vizsgálatára*

***Input**: Egy CG=(V(CG), E(CG)) vezérlésfolyam-gráf*

***Output**: $C = \{c_1, c_2, \ldots, c_n\}$, a függvények osztályainak halmaza*

```
1   N := FüggvényekSzáma()
2   max_fin := Maxfin(N)
3   smin := MinimálisOsztályméret(N)
4   F :=V(CG) //A további vizsgálandó függvények halmaza
5   For ∀f ∈ F do
6   |     c := Vizsgál(f, {})
7   |     if |c| ≥ smin then
8   |     |     Hozzáad(C, c)
9   |     end
10  |     else do
11  |     |     For ∀g ∈ c do
12  |     |     |     Hozzáad(F, g)
13  |     |     end
14  |     end
15  end
16  return C
```

***Vizsgál** eljárás*

***Input:** $f \in V(CG)$ függvény*

***Input:** c, a vizsgált lánc függvényeinek bővítendő halmaza*

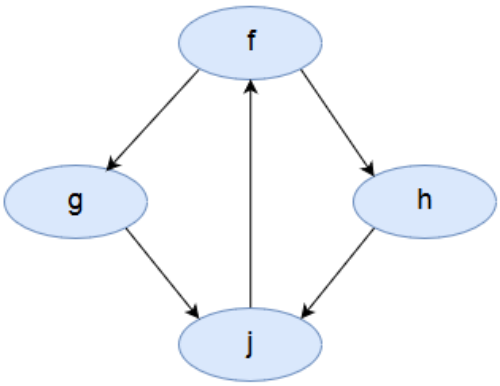
***Output:** $c=\{f_1, f_2, \ldots, f_k\}$ a lánc osztályba szervezett függvényeinek halmaza*

```
1   Kivesz(F, f)
2   fin := BejövőÉlekSzáma(f)
3   if fin > max_fin then
4   |     return {}
5   end
6   Hozzáad(c, f)
7   For ∀g ∈ Leszármazottak(f) do
8   |     if g ∈ F then
9   |     |     c = Vizsgál(g, c)
10  |     end
11  end
12  return c
```

Egy csomópont esetleg több, különböző osztályhoz is hozzáadható lehet annak függvényében, hogy milyen sorrendben járjuk be a csomópontokat. Az algoritmus mohó módon működik, mert ha egy adott csomóponthoz talál osztályba szervezési lehetőséget, akkor azt azonnal megpróbálja végrehajtani. Ennek előnye, hogy ha létezik ilyen, akkor mindig meg fog találni egy lehetőséget. Hátránya azonban, hogy nincs garancia arra, hogy a legoptimálisabb eredményt kapjuk, ugyanis előfordulhat, hogy ha egy adott lépésben egy csomópontot nem adnánk hozzá az épülő osztályhoz, akkor azzal később egy jobb szerkezetet lehetne kialakítani.

Az algoritmus nem látogatja meg gyökérként azokat a csomópontokat, amiket egyszer leszármazottként már megvizsgált és azt tapasztalta, hogy a bejövő fokszáma túl nagy. Így felesleges vizsgálatokat nem végez.

Előfordulhat az is, hogy a CG-ben kör található. Ez jelen kontextusban azt is jelentheti, hogy a vizsgált alkalmazásnak létezik olyan vezérlési útja, ahol egy végtelen hívási láncba kerül. Egy ilyen esetet mutat be a ***8. ábra***, ahol két kör is található: az ***f-g-j-f***, illetve az ***f-h-j-f*** vezérlési úton.



8. ábra: Kör a CG-ben.

Ez azért jelent problémát az algoritmus futása során, mert ha valóban végtelen hívásról van szó, akkor a függvények leszármazottjainak vizsgálata során maga az algoritmus is végtelen futásba fog kerülni. Az algoritmus ezért figyel arra, hogy a **Vizsgál** eljárás során egy csomópontot többször ne látogasson meg.

Lássuk most egy egyszerű példán az algoritmus futását. Tekintsük példaként a ***7. ábra*** hívási gráfját és az egyszerűség kedvéért feltételezzük, hogy a programban található összes függvény bemenete lett a hívási láncok vizsgálatának. A csomópontok bejárása

véletlen sorrendben történik. Tegyük fel, hogy az algoritmus például ***f-g-h-j-l-m-k*** sorrendben látogatja meg a függvényeket gyökérként. Első lépésként az algoritmus kiszámítja a határértékeket:

- $max f_{in} = \lceil 0{,}2 * 7 \rceil = 2$
- $s_{min} = \lceil 0{,}1 * 7 \rceil + 1 = 2$

Ezután a csomópontok bejárása következik. Az algoritmus futását a ***3. táblázat*** mutatja be. Egy adott sor alatti vastagított elválasztó vonal azt jelzi, hogy egy hívási lánc vizsgálatát az algoritmus befejezte. Ekkor a kialakítandó osztály cellában lévő ✓ azt jelenti, hogy az osztály mérete elegendő a létrehozáshoz, az ✘ pedig azt, hogy nem.

3. táblázat: Az algoritmus futása.

Lépészám	*Vizsgált csomópont*	*Bejövő fokszám*	*Hátralévő csomópontok*	*Kialakítandó osztály*
1.	*f*	*0*	*{g, h, j, l, m, k}*	*{f}*
2.	*g*	*1*	*{h, j, l, m, k}*	*{f, g}*
3.	*j*	*5*	*{h, l, m, k}*	*{f, g}*
4.	*l*	*1*	*{h, m, k}*	*{f, g, l}*
5.	*m*	*1*	*{h, k}*	*{f, g, l, m}*
6.	*h*	*1*	*{k}*	*{f, g, l, m, h}* ✓
7.	*k*	*1*	*{ }*	*{k}* ✘

Az első meglátogatott függvény ***f***. Azt látjuk, hogy $f_{in} = 0$, tehát megvizsgáljuk a gyerekeit is. Ekkor $g_{in} = 1$, tehát tovább mehetünk. A j csomópontra viszont $j_{in} = 5$, így ezt a részfát nem vizsgáljuk tovább. Megyünk tovább ***g*** következő leszármazottával, $l_{in} = 1$, tehát ***l*** gyerekei következnek. Mivel ***j***-t már egyszer megvizsgáltuk, és további gyerek nincs, ***m*** következik, aki szintén csak ***j***-t hívja. Így a ***g*** oldali részfát bejártuk, jöhet a ***h*** oldali. Itt $h_{in} = 1$, leszármazottja pedig megint csak ***j***, így ***f*** vizsgálata befejeződik. Az összegyűjtött függvények: ***f, g, l, m, h***. Mivel $5 > s_{min}$, ezért ebből készítünk egy osztályt. Mivel j_{in} már egyszer túl nagynak bizonyult, így azt nem is látogatjuk meg többé. Az összes többi függvénnyel már végeztünk, így ***k*** következik, akinek viszont nincs több leszármazottja, így az algoritmus futása itt véget ér. A ***k*** csomópont osztályjelölt ugyan, de más függvény nem került mellé, így $1 < s_{min}$ miatt ebből nem lesz osztály. Egy osztályt sikerült tehát kialakítani: ***f, g, l, m, h*** tagokkal.

5.2.4 Az öröklés kérdése

Természetesen egy jó objektumorientált alkalmazás az OO paradigmák közül az öröklést is (és implicit a polimorfizmust is) kihasználja. Procedurális kódok esetében azonban egyáltalán nem áll rendelkezésre semmilyen információ, amely alapján öröklést tudnánk bevezetni a modell osztályai között anélkül, hogy a *Liskov elvet*[9] megsértenénk. Egy helyesen megtervezett OO modell ugyanis az öröklést a viselkedés kiterjesztésére kell, hogy használja, nem az adatok újrahasznosítására. A C kód vizsgálata során legfeljebb adatokat újrahasznosító örökléseket tudnánk detektálni, mert az alkalmazás szemantikájának mélyebb ismerete nem áll rendelkezésre. Ezért az öröklés bevezetése automata módon nem megoldható feladat.

5.3 API hívások transzformációja

A procedurális kód természetesen tartalmaz olyan függvényhívásokat is, melyek az adott nyelv beépített könyvtári függvényeit hívják. Ilyenek lehetnek pl. a matematikai függvények, a sztenderd bemenet írása/olvasása, sztringműveletek vagy a fájlkezelés. Ezek automatikus leképzése messze nem triviális feladat, ezért szükséges egy módszer, amellyel a konverziót hatékonyan le tudjuk írni.

Első megoldásként valamilyen konfigurációs fájlra gondolhatnánk, pl. Xml-re [28], ez azonban a probléma potenciális komplexitása miatt nagyon bonyolult és emberi szemnek olvashatatlan lenne.

Ezért ehelyett erre a problémára a következő módszert használhatjuk. Ahhoz, hogy meg tudjuk közelíteni, szükségünk van egy adatszerkezetre, ami hatékonyan leírja, hogy az egyes függvényeknek mi a megfelelő képe a célnyelvben. Ennek érdekében egy olyan egyszerű szakterületi nyelvet fejlesztettem ki az ***Xtext*** segítségével, amely az egyszerű alapesetekre hatékony megoldást ad. Később ez kibővíthető, hogy a bonyolult eseteket – pl. fájlkezelés, változó paraméterű függvények, kifejezések paraméterként szerepeltetése – is kezelje. A nyelvünk tehát a következőképpen néz ki:

```
Model:
     transformations+=Transformation*
;
```

[9] https://stackify.com/solid-design-liskov-substitution-principle/

```
Transformation:
	source=Function '=>' target=Target
;

Target:
	static?='static'? (prefix=FunctionPrefix)? function = Function
;

FunctionPrefix:
	prefixes+=ID ('.' prefixes+=ID)* '.'
;

Function:
	name=ID '(' parameters+=Parameter* ')'
;

Parameter:
	name=ID
;
```

A nyelv függvények transzformációját írja le. Egy transzformáció két függvényből áll, amelyeket a ’=>’ jel választ el. A függvényeknek van egy egyedi azonosítójuk (nevük), amelyet zárójelek között a szintén egyedi névvel ellátott paraméterek felsorolása követ. Egy transzformáció bal oldala egy ilyen függvényt tartalmaz. A jobb oldal ezzel annyiban van kiegészítve, hogy a ’.’ operátorokkal tetszőleges mélységű hierarchiát tudunk megadni a célfüggvény azonosítása érdekében. Erre azért van szükség, mert az objektumorientált nyelvekben a függvényekre osztályok (esetleg statikus) metódusaiként hivatkozunk.

A célfüggvényben azt is megadhatjuk, hogy a prefixumot statikus osztályként értelmezzük-e, vagy egy olyan osztályként, amelyből példányt kell létrehoznunk és azon meghívni a megadott szignatúrájú metódust. Az utóbbi eset természetesen jóval bonyolultabb, ekkor ugyanis a példányosításhoz szükségünk lehet paraméterekre is. Az sem kizárható továbbá, hogy egy híváshoz több, esetleg különböző osztályok példányai szükségesek. Ezek közül a nyelv jelenleg csak az alapesetet kezeli, vagyis egy példányt, paraméter nélkül.

Az ***Xtext*** ebből a leírásból egyebek mellett automatikusan parsert generál, mely a ***2.2.*** fejezetben ismertetetteket végzi el. Generál továbbá egy szerkesztőt is Eclipse pluginként. Ebben a szerkesztőben lehet elkészíteni ezek után a megfelelő szkriptet, amelyben leírhatjuk a transzformációkat.

Adott pl. a következő szkript, amely néhány egyszerű C forrásnyelvű függvény Java nyelvű megfeleltetését adja meg:

```
abs(x)          =>   static Math.abs(x)
time(NULL)      =>   static System.currentTimeMillis()
rand()          =>   Random.nextInt()
```

A leképzések megadásában tetszőlegesen változtathatjuk a forrásfüggvény paramétereit. Átvehetjük értelemszerűen, megváltoztathatjuk a sorrendüket, illetve akár el is hagyhatjuk őket, ha a célnyelvű függvény szignatúrája nem igényli azokat. Láthatjuk azt is, hogy az első két esetben statikusként jelöltük meg a cél prefixumát. Az utolsó esetben viszont nem: ezzel jelezzük, hogy a feldolgozás során ebből egy példányt kell majd létrehozni.

Az Xtext ebből a szkriptből képes modellt generálni, amit programozottan kinyerhetünk és felhasználhatunk. Ez a modell lesz az az adatszerkezet, amely megadja, hogy az egyes függvényeknek mi legyen a transzformált megfelelőjük.

A konverzió tehát úgy működik, hogy az előző lépésekben létrehozott OO modell függvényhívásainak bejárása közben az Xtext által generált transzformációs modellt bejárva megkeressük, hogy az adott hívásnak mi a transzformáltja, majd ennek megfelelően átalakítjuk. Ez, ahogy az OO modell felépítése során is láttuk, szintén egy M2M transzformáció lesz.

6 A transpiler

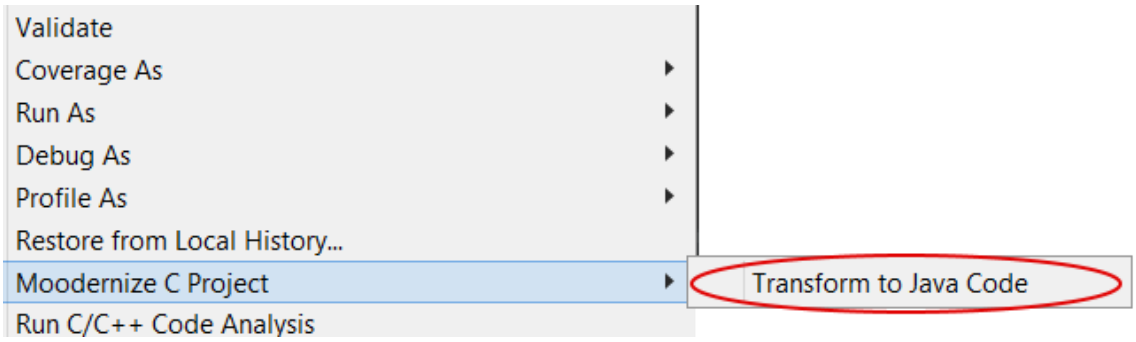
Ebben a fejezetben a korábbiakban leírt módszert megvalósító és szemléltető transpilert mutatom be.

Bemutatom az eszköz felépítését, kitérve az egyes modulok szerepére. Ismertetem a két nyelv közötti konverzió érdekesebb, bonyolultabb kérdéseit, a tesztelhetőség nehézségeit és annak egy lehetséges megoldását. Megnézzük azt is, a transpiler jelenleg milyen nyelvi elemeket nem kezel még vagy kezel helytelenül.

6.1 A megvalósítás kontextusa

A transpiler Java nyelven, Eclipse pluginként lett megvalósítva. Futtatásához tehát Eclipse fejlesztőkörnyezetre, és JavaSE-1.8[10]-ra van szükség. Ezen felül a sikeres futáshoz elérhetőnek kell lennie a CDT és JDT technológiáknak is. Ezek pl. az Eclipse beépített „Install New Software" funkciójával telepíthetők.

A transzformáció indítását a plugin telepítése után az Eclipse-ben C projektek kontextus menüjében érhetjük el. Ezt a ***9. ábra*** mutatja.



9. ábra: A transzformáció indítása.

Minden .c kiterjesztésű forrásfájlt és .h headert az Eclipse projekten belül source mappákban kell elhelyezni, hogy a transzformáció felismerje őket forrásállományként.

[10] http://www.oracle.com/technetwork/java/javase/downloads/index.html

Ha nem ilyen mappába (hanem pl. sima könyvtárba) helyezzük őket, akkor a transzformátor figyelmen kívül fogja azokat hagyni.

6.2 A transpiler felépítése

A transpiler felépítése a transzformáció módszertanához igazodik.

1. Főmodul (mag)
2. Transzformátor
3. OOGen metamodell
4. Kódgenerátor

A 2-4. modulok megfeleltethetőek a módszertan adott lépéseinek. Megjegyzendő, hogy különálló elemző (parser) modulra nincsen szükség, mert a CDT elemzője rendelkezésre áll a C projekt feldolgozására.

6.2.1 A transzformáció magva

Az első komponens a transzformáció összefogásáért felelős.

Ez a modul biztosítja az eszköz számára az Eclipse pluginként való működést. C projektek kontextusmenüjét kiterjeszti egy „Moodernize C Project” funkcióval, amely elindítja az alkalmazást. Ez a modul vezényeli le a többi komponens feladatait. A CDT-t felhasználva programozottan feldolgoztatja a forrásprojektet, meghívja a transzformátort a kapott szintaxisfákra, majd a JDT és a Java nyelvre fordító kódgenerátor felhasználásával generálja az alkalmazás kódját és azt egy Eclipse projektbe szervezi.

Természetesen nincs semmilyen megkötés, hogy csak Eclipse-ben karbantartható a továbbiakban a projekt. Az elkészült forráskódot akármilyen más alkalmas környezetben is lehet használni.

6.2.1 A transzformátor

A második komponens maga a transzformátor. Ez AST-k halmazát várja és eredményül az elkészült objektumorientált modellt adja. Itt történik minden, ami a modell kialakításához szükséges.

A C kód minden elemének eléréséhez a CDT típusos API-t kínál. A kódot szintaxisfák formájában kapjuk, ezért szükség van egy hatékony módszerre, amivel bejárhatjuk ezeket a fákat. A CDT-ben erre a *Visitor*[11] tervezési minta használható.

Első lépésként tehát a transzformátor ezt használva bejárja a kapott AST-ket. Ha fordítási hibát talál, azt azonnal kivételben jelzi a hívójának, aki abortálni fogja a transzformációt és jelzi a hiba jelenlétét a felhasználó számára.

Ha nem található fordítási hiba a kódban, akkor a fákat bejárva a transzformátor összegyűjti az alkalmazás entitásait: struktúrákat, globális változókat, függvényeket és azok definícióit. Minden elemhez egy konverter segítségével kialakítja az ő megfelelő Java-beli képét, így felépítve az OO modell alapját.

Ezután a módszertan alkalmazása következik, a transzformátor az ***5***. fejezetben leírtakat végzi el az összegyűjtött entitásokon. Ennek végeztével elkészül a modell, ami az objektumorientált programot írja le.

A transzformátor működése során esetenként felléphetnek nem fordítási hibák is. Ilyen pl. ha egy nem szupportált vagy még nem megvalósított elemet talál a kódban. Ekkor a transzformátor teljes működése nem áll le egyetlen hiba miatt, hanem a hibás elemhez egy üres kifejezést/utasítást fog generálni.

6.2.1 OOGen – az alkalmazás metamodellje

Az objektumorientált alkalmazás modellje egy EMF metamodell példánya, amely egy nyelv-független, objektumorientált programot ír le. A modell tartalmazza az OO nyelvek közös vonásait, illetve ezen felül támogatást nyújt nyelvi különlegességek kezelésére is. A metamodell tehát a nyelvi elemek és képességek unióját támogatja, nem a metszetét.

A metamodell reprezentációja praktikus okokból ábrákon nehezen bemutatható, mert túl sok elemet tartalmaz. A metamodell teljes jelenlegi állapota elérhető és megtekinthető a ***6.6.*** alfejezetben található github linket követve.

[11] https://www.tutorialspoint.com/design_pattern/visitor_pattern.htm

6.2.2 A kódgenerátor

A létrejött OO modellt fel is kell dolgozni, abból kódot kell készíteni. Ezt a feladatot látja el a kódgenerátor, amely a kapott modellelemekből kódot készít az adott célnyelven. A kódgenerátor tehát nyelvfüggő modul, minden támogatni kívánt célnyelvhez el kell készíteni egy kódgenerátort is.

A generátort Xtend nyelven valósítottam meg Java célnyelvre. Az Xtend [29] Java-ra forduló, sablonalapú nyelv, mely nagyon jól illeszkedik EMF modellekből való kódgeneráláshoz. A nyelv *dispatch*[12] mechanizmusa (*polymorphic method invocation*) kiváltja a típus szerinti bonyolult, rekurzív esetszétválasztásokat. Ezzel tovább gyorsítja a fejlesztést és átláthatóbbá teszi a generátor forráskódját. A sablonokban egyszerűen megadhatjuk a generált kód felépítését.

A generátor osztályt vagy enumerációt vár és elkészíti sztringként annak forráskód reprezentációját. Egy meglévő modell alapján tehát úgy tudunk kódot készíteni, hogy bejárjuk annak minden osztályát és a kódgenerátor segítségével elkészítjük a hozzájuk tartozó forráskódot sztringként. Ezek lesznek az egyes osztályok tartalmai, amelyet már a JDT API segítségével fel tudunk használni a konkrét fájlok generálásához.

6.3 Szisztematikus funkcionális tesztelés

A transpiler automatikus tesztelése bonyolult feladat. Ennek oka, hogy az eszköz bemenetként egy projektet vár, amit feldolgoz. Így állítja elő az AST-ket, amelyekkel a transzformátor dolgozni tud. Kérdés, hogy AST-ket egy automatikus tesztkörnyezetben hogyan tudunk előállítani.

Elméletben felépíthetnénk programozottan egy virtuális fájlrendszert, vagy akár készíthetnénk AST-ket kézzel is. Ez azonban nagyon bonyolult, mechanikus és hibaérzékeny feladat volna. Ennél van egyszerűbb megoldás is. A CDT ugyanis képes sztringből is AST-ket készíteni. A tesztelés tehát *JUnit*[13] [30] tesztekbe szervezhető a következő módon:

---

[12] https://www.eclipse.org/xtend/documentation/

[13] https://junit.org/junit5/

1. Megírjuk a tesztelendő kódrészletet sztringként.
2. A CDT-t használva elkészítjük az AST-ket a sztringből.
3. Az AST-ket átadjuk egy transzformátornak, ami elkészíti a hozzá tartozó modellt.
4. Az elkészült modell elemeit már vizsgálhatjuk, hogy pontosan azok az elemek jöttek-e létre, amelyeknek az adott kód alapján létre kellett jönniük.

Nézzünk egy példát ezek alapján egy teszt felépítésére:

```
@Test
    public void prefixDecrement_transformToOOPrefixDecrementExpression() {
     StringBuilder sourceCode = new StringBuilder();
     sourceCode.append("int globalInt;");
     sourceCode.append("void someFunction() {");
     sourceCode.append("  --globalInt;");
     sourceCode.append("}");

     OOModel model = getModelBySourceCode(sourceCode.toString());

     OOMethod someFunction = getDefaultClass(model).getMethods().get(0);
     OOStatement statement = someFunction.getStatements().get(0);
     Assert.assertTrue(statement instanceof OOPrefixDecrementExpression);
}
```

A teszt definiálja a tesztelendő kódrészletet, majd elkészíti a modellt. Ezután megkeresi a modellben a releváns részt és ellenőrzi, hogy a prefix dekremens operátornak megfelelő kifejezés jött-e létre.

Megjegyzendő, hogy a tesztelés ilyen formája leginkább *integrációs*[14] tesztnek minősíthető, mert a transzformátor egészének kimenetét vizsgálja. Az egységtesztek szintjén le kellene menni egészen az egyes utasításokat transzformáló konverterekig, azonban a transzformátor kimenetének helyességét explicit vizsgálni sokkal hasznosabb visszajelzést ad a fejlesztés során. A tesztesetek ugyanis eléggé izoláltak ahhoz, hogy az esetleges hibák könnyen azonosíthatók legyenek.

6.4 Nyelvi elemek konverziója

A C és Java nyelvek közötti konverzió tartalmaz néhány kulcskérdést és bonyolultabb problémakört, amelyeket kezelni kell. Ezeket a problémákat tekintjük most át részletesen.

[14] http://softwaretestingfundamentals.com/integration-testing/

6.4.1 Típuskonverzió

A transzformáció során le kell képezni a C típusait a célnyelv típusaira. Az alapvető típusok megfeleltetését a ***4. táblázat*** szemlélteti:

4. táblázat: A C típusainak konverziója Java típusokra.

C típus	*Java típus*
int, signed int, unsigned int, unsigned short, unsigned short int, short, short int, signed short, signed short int, signed, unsigned	***int***
long, long int, signed long, signed long int, unsigned long, long int, long long, long long int, signed long long, signed long long int, unsigned long long, unsigned long long int	***long***
char, unsigned char, signed char	***byte***
float, double, long double	***double***

Ez az összes lehetséges C-beli alapvető típust összefoglalja. Az egyes típusok konvertálásakor ügyelni kell arra, hogy a céltípus számábrázolási tartományának tartalmaznia kell a C-beli típus tartományának egészét. Ez az oka annak, hogy a C „*char*” típusának képe a „*byte*”, nem a Java-s „*char*”.

A *void* típus a C nyelvben több kontextusban is előfordulhat, ezek más-más kezelési módot igényelnek:

- Ha függvény visszatérési értékéről van szó, akkor a *void* képe *void* lesz.
- Ha függvény paramétereként szerepel, az C-ben azt jelenti, hogy nincs paramétere a függvénynek. Ekkor nincs mit konvertálni, a leképzett metódusnak egyszerűen nem lesz paramétere.
- Ha *void** szerepel – azaz egy olyan pointer, ami adatra hivatkozik, de az adat típusáról nem ad információt – akkor a konvertált típus *Object* lesz, mert ez az összes létező Java osztálynak őse, így pontosan kifejezi a *void** szerepét.

6.4.2 Pointerek leképzése

A C pointereinek leképzése Java kódra nem triviális probléma. Egy pointer több lehetséges kontextusban is előfordulhat, amelyeket máshogy kell kezelni. A következő két esetet különböztethetjük meg:

1. Pointer „hagyományos” pointerként használva. Lehet dereferálni, pointer aritmetikát végezni vele, címképző operátort használni, más pointert értékül adni neki.
2. Pointer tömbként használva. A C nyelvben a tömbök a legelső elemre mutató pointerként értelmezettek.

A két probléma között fontos különbséget tenni, mert teljesen máshogy kell őket transzformálni. Az első esetben egy pointer képe egy sima Java-s változó (referencia), amely az adott típusú adatra referál. A dereferáló és címképző operátorokat el kell hagyni. A második esetben valójában egy tömbről van szó, melyet tömbként kell leképezni, nem egyszerű változóként.

Ezt a problémát a következő módszerrel közelíthetjük meg. Tekintsünk alapértelmezetten minden pointert tömbnek. A pointer használatát vizsgálva keressünk arra utaló jelet, hogy azt mégsem tömbként kell kezelni, hanem hagyományos pointerként. Erre a pointeraritmetika detektálása nyújt megoldást.

A preinkremens, posztinkremens, predekremens, posztdekremens és dereferáló operátorokat kell figyelnünk. Ha azt látjuk, hogy addig tömbnek feltételezett változón ilyen műveleteket végzünk, akkor retroaktívan módosítanunk kell az adott változó típusát. A változó ráadásul többféle eredetű lehet: globális változó, lokális változó, függvényparaméter, tagváltozó.

A tagváltozó és a globális változó közvetlenül kikereshető és a módosítás helyben elvégezhető, de a másik két esetben „visszafelé” kellene látnunk a függvényben a hozzájuk tartozó deklaráció megtalálásához. Erre a problémára használhatunk *szimbólumtáblákat*[15]. Egy szimbólumtábla azonosítók (pl. egy változó neve) előfordulásával és deklarációjával kapcsolatos információkat tartalmaz. Ebben az esetben elég, ha minden pointer típusú paraméter és lokális változó esetén egy szimbólumtáblában nyilvántartunk egy referenciát a változó deklarációjára. Ezt felhasználva egy tetszőleges változónév alapján kikereshető annak deklarációja és átalakítható típusa.

[15] https://en.wikipedia.org/wiki/Symbol_table

A szimbólumtáblát elég addig memóriában tartani, amíg az adott függvénydefinícióban vagyunk. Amint új függvénybe lépünk, az előző tartalmát törölhetjük, mert nem lesz már rá szükségünk.

Figyelünk kell természetesen arra is, hogy a névütközések ne okozzanak problémát. Ez szerencsére esetünkben automatikusan teljesül. A C nyelv hatókörkezelése [31] a legtöbb nyelvhez hasonlóan belülről kifelé halad. Ez azt jelenti, hogy ha létezik pl. egy „x” nevű lokális és globális változó is, akkor a függvényben az „x” azonosító a lokális változóra hivatkozik. Ezért a szimbólumtáblába mindig a helyes azonosító-deklaráció páros kerül.

Speciális kezelést kívánnak továbbá a memóriakezelő függvények. Az explicit felszabadítást természetesen el kell hagyni, azt a *garbage* collectorra kell bízni. A memóriafoglalás művelete szerencsére leképezhető objektumorientált nyelvekre is. A foglalás paramétereit felhasználva el tudjuk dönteni, hogy mennyi elemet kell foglalnunk és az értékül kapott pointer típusát látva a foglalandó elem típusa is rendelkezésre áll.

Ezért pl. egy *malloc* hívás a következőképpen transzformálható:

```
int *p1 = (int *) malloc (100 * sizeof(int));
int *p2 = (int *) malloc (40);
```

```
int[] p = new int[100];
int[] p = new int[10];
```

A második hívás konvertálásához felhasználandó, hogy az *int* típus 4 bájt méretű, ezért 40 / 4 = 10 elemet kell foglalnunk. Általános esetben, ha ***M*** jelöli a kívánt memóriamennyiséget és ***T*** a kért típus méretét bájtokban mérve, akkor összesen $\lceil \mathbf{M} / \mathbf{T} \rceil$ darab elemet kell foglalnunk.

6.4.3 További konverziós problémák

A nyelvi elemek konverziója során felmerül néhány további nem triviális probléma, melyeket kezelni kell. Ezek a következők:

1. Booleankezelés
2. Kommentek leképzése
3. Referenciagenerálás getter híváshoz

4. Paraméter függvényhívásainak lecserélése *this* kulcsszóval

A C **boolean kezelésével** az a probléma, hogy nincs explicit boolean típus a nyelvi elemek szintjén. Helyette egész számok használhatóak ilyen célra: a logikai hamisnak a 0 felel meg, minden más egész szám logikai igazként van értelmezve, ha logikai kifejezésben szerepel. Ezért ha a program valamely pontján egész számot találunk, akkor meg kell vizsgálnunk, hogy az milyen kontextusban fordul elő. Ha logikai kifejezésben van, akkor át kell alakítani az értékének megfelelő boolean értékre.

Mindez sokkal bonyolultabb kontextusban is előfordulhat, pl. egy függvényhívás ilyen. Ekkor az integerként átadott értékről a függvény definíciójának vizsgálata nélkül nem tudjuk eldönteni, hogy integerként vagy booleanként van használva. Ráadásul ugyanez a paraméter akár híváson belül is felhasználható további híváshoz, ami mélyrekurzív vizsgálathoz vezet. A transpiler az esetek többségét helyesen tudja kezelni, a még nem kezelt esetekről a ***6.5***. fejezetben olvashatunk.

A kódban előforduló **kommentek leképzése** ránézésre nem tűnik bonyolult feladatnak, hiszen kommentek szinte minden nyelvben vannak és többnyire uniform módon használhatóak. A probléma itt a kommentek elhelyezéséből adódik. A sorszám, ahol a komment található, ugyanis nem elegendő információ a komment elhelyezéséhez. Ez azért van így, mert a transzformáció során a sorok száma változik, a sorszámozás elcsúszik egymástól.

Ezt a kérdést a transpiler a következő módon oldja meg. A kommenteket nem sorszám, hanem kontextus alapján helyezi el a modellben. Ez azt jelenti, hogy minden olyan nyelvi elemhez kommenteket csatolhatunk, amelyek elé vagy után kommentet tehetünk a kódba. Ezek az elemek az osztályok, enumerációk, utasítások, kifejezések, függvénydeklarációk és a tagváltozók. Egy komment detektálásakor tehát meg kell nézni, hogy a felsorolt elemek közül melyikhez található a legközelebb és az annak az elemnek megfelelő modellelemhez kell csatolni azt. Az, hogy az elem elé vagy után kell elhelyezni, a CDT által előállított sorszám (és azon belüli offset) összehasonlításával egyértelműen eldönthető.

A kommentek begyűjtésekor az eszköz teljesítményének figyelembevétele miatt a transpiler csak a kommentek közvetlen közelét vizsgálja meg (elejétől és végétől számítva + / - 5 sor), így nagyságrendekkel gyorsabban képes elvégezni a tulajdonosok

megtalálását. A közvetlenül egymás után írt egysoros kommenteket át kell ezért alakítani blokk kommentté, hogy az 5 soros limit miatt ne hagyjunk ki egyetlen kommentet sem.

A módszertan ismertetésénél említettem, hogy az osztályok tagváltozóinak elérését **getter** / setter hívásokra transzformálom. Ezzel azonban az a probléma, hogy közvetlenül egy getterhíváson keresztül a tagváltozó nem módosítható. A hívástól kapott referenciát el kell tárolnunk, a továbbiakban azon keresztül érhetjük el a változót. Ezért egy tagváltozó referenciájának képe az eddigiekkel ellentétben nem feltétlenül egy egyszerű változó referencia kifejezés. Szükséges lehet egy új változót létrehoznunk, amelyben a referenciát fogjuk tárolni. Ráadásul figyelembe kell vennünk azt is, hogy az adott példány adott getterhívásához hoztunk-e már létre referenciát. Ha igen, akkor azt kell felhasználnunk, ellenkező esetben minden híváskor egy új változót generálnánk, ami hiba lenne.

Ennek a problémának a megoldása nagyon hasonló a pointerek leképzésénél látott tömb típusok módosításának esetéhez. Ugyanúgy egy *szimbólumtáblát* használhatunk, csak most azt kell eltárolnunk, hogy az adott hatókörön belül hoztunk-e már létre referenciát adott azonosítóval rendelkező változó adott azonosítóval rendelkező tagváltozójának eléréséhez. Ha igen, akkor azt használjuk, ha nem, akkor létre is hozzuk. A létrehozáskor a változó nevéhez egy „generated” prefixum hozzáfűzésével biztosítjuk a névütközések elkerülését[16].

Az **utolsó probléma** a használt módszertan miatt keletkezik. Az ***5.2.2.*** fejezetben bemutatott paramétervizsgálat esetében ugyanis ha struktúra típusú paraméter alapján osztályba szerveztünk egy metódust, akkor azt a paramétert el kell hagyni és a rajta történő hívásokat le kell cserélni az adott példány hívásaira. Ez volt a koncepció lényege és emiatt tehát ebben a függvényben a paraméteren végzett minden hívást le kell cserélni az adott példány hívásaira.

Ehhez tudnunk kell, hogy egy adott függvényből elhagytunk-e paramétert vagy nem. Mivel a függvények paraméter-és visszatérési érték vizsgálata megelőzi a függvények

[16] Valójában tökéletesen biztosak nem lehetünk abban, hogy véletlenül nem létezik-e egy olyan nevű változó is, amely pont a “generated” prefixumból és a változó nagybetűsített eredeti nevéből áll. Erről meggyőződhetnénk szintén szimbólumtáblát használva, azonban ennek az esélye és a hiba okozta “kár” annyira minimális, hogy nem éri meg ilyen vizsgálatokat végezni.

definíciójának transzformációját, ezért a vizsgálatkor (mivel akkor töröljük a paramétert) a paraméter törlését megjegyezzük. Ezt felhasználva a definíció transzformálásakor ha egy adott hívás tulajdonosa (amin a metódust meghívjuk – szemléletesen X.foo() esetén X a tulajdonos) megegyezik az elhagyott paraméterrel, akkor lecseréljük a tulajdonost *this* kulcsszóra.

6.5 A transpiler jelenlegi hiányosságai, limitációi

A transpiler néhány nyelvi elemet és speciális konverziós esetet még nem kezel helyesen.

A nyelvi elemek közül a preprocesszor utasítások és a függvénypointerek transzformációja még nincs megvalósítva. Ez nem nehézségek miatt van így, egyszerűen még nem kerültek sorra. Hexadecimális és normálalakban megadott literálisok konverziója sem támogatott még. Itt fontos ügyelni arra, hogy pl. a normálalakban megadott számok hatalmasak lehetnek, így a konvencionális típusokban valószínűleg nem férnek el.

Hasonlóan nem készült még el pl. a *sizeof* operátor, illetve a NULL pointerek feloldása. A null pointerből a transzformátor jelenleg „(Object) 0”-t generál, mert a NULL nem más, mint egy makródefiníció a „(void *) 0” kifejezésre, amit viszont a CDT már csak feloldott formájában lát. Amíg ez nincs kiküszöbölve, a fejlesztőkörnyezet „replace” funkciójával kell kézzel lecserélni az „(Object) 0” találatokat „null”-ra.

A transpiler nem támogatja a *union* használatát, mert az egy ritkán előforduló, meglehetősen ellenjavalt nyelvi elem. Ha a kód ilyet tartalmaz, azt transzformáció előtt kézzel át kell alakítani ekvivalens, *union*-mentes kódra.

Ezeken kívül természetesen a helytelen bemeneti C kód is okozhat helytelen működést. A transzformátor fel van készítve arra, hogy a hibás, fel nem ismert vagy még nem támogatott elemeket üres kifejezésekkel/utasításokkal generálja. Így nem kell megszakítani a teljes folyamatot ilyen esetek miatt, azokat a transpiler futásának vége után kézzel kijavíthatjuk.

Jellemzően ilyen problémákat okoz, ha egy struktúrának nincs meg a definíciója (mert pl. külső könyvtárból származik). Ez az ilyen típusú változók adatelérésénél gondot okoz,

mert az adott változó típusának megfelelő osztály nem fog szerepelni a transzformátor által nyilvántartott osztályok között és ez bizonyos műveleteknél hibát okozhat.

A booleankezelés konvertálásának van két komplexebb esete, melyet még nem kezel helyesen az eszköz. Ezek egyike a korábbiakban is említett függvényhívások esete, a másik struktúrák inicializáló listái (amelyek konstruktorrá fordulnak). Itt a hívásokhoz hasonlóan szemantikai analízis segítségével fel kellene deríteni, hogy booleanként vagy integerként használja-e az adott típus az adott tagját. Itt megoldást jelenthetne az is, ha a felhasználót bevonjuk a transzformációba. Megkérhetjük arra, hogy mondja meg ő, integerként vagy booleanként van-e használva az adott változó.

A pointerek leképzése sem hiánymentes, szükséges lehet további módszerekkel finomítani a megadott két eset megkülönböztetését. A transzformátor a megadott módszer mellett még tömbnek ítélhet meg olyan pointert, amit egyszerű változóként kellene kezelnie.

A kommentek konvertálásának speciális esete, amely nincs kezelve, az üres blokkokban található komment. Erre egy egyszerű szemléletes példa a következő kódrészlet:

```
if (condition) {
     // Comment in empty block
}
```

Ekkor, mivel nincs a blokkon belül semmi, amihez csatolhatnánk a kommentet, az a legközelebbi elemhez, vagyis az *if* logikai feltételéhez fog csatolódni. Így, bár a jó hely közvetlen közelébe, mégis rossz helyre kerül. Ezen felül, a korábbiaknak megfelelően, ha egy komment közvetlen közelében nincsen semmi (elejétől és végétől számítva + / - 5 sor), akkor a komment elveszik. Ez azonban nagyon ritkán fordul elő és az esetek többségében nem okoz problémát.

A transpiler jelenleg tömbökből tömböket készít, amelyek helyett modern nyelvekben kollekciókat (halmazok, listák) szokás használni. A kollekciók generálása a jövőben könnyedén kivitelezhető, de ezt egy tetszőleges refaktoráló eszköz is el tudja végezni a tömböket használó generált kódon.

6.6 Elérhetőség

Az eszköz komponensei a következő URL-eken érhetők el:

- https://github.com/gaborbsd/Moodernize/tree/master
- https://github.com/gaborbsd/OOGen/tree/master

A transpiler működéséhez a következő projekteket kell letölteni:

1. **hu.bme.aut.moodernize.c2j**: A transzformátor modulja.
2. **hu.bme.aut.moodernize.ui**: Az eszköz Eclipse-pluginként való működését biztosítja. Vezérli a transzformáció folyamatát.
3. **hu.bme.aut.moodernize.meta**: Az OOGen metamodell definíciója.
4. **hu.bme.aut.oogen.general**: Az OOGen kódgenerátoraihoz szükséges közös interfészek és funkcionalitások gyűjteménye.
5. **hu.bme.aut.oogen.java**: Az OOGen Java célnyelvre fordító kódgenerátora.

A futtatáshoz importáljuk be a megadott projekteket, majd Runtime Eclipseben futtassuk pl. a *hu.bme.aut.moodernize.ui* projektet. Ezt követően a ***6.1***. fejezetben leírtak szerint kell eljárnunk.

7 Esettanulmányok

Ebben a fejezetben néhány nyílt-forráskódú projekt transzformációját tekintjük át. Megnézzük, milyen bemenetből milyen kimenetet generál a transpiler, illetve azonosítjuk a sokszor előforduló hibákat a kimenetben. A fejezet célja tehát nem az eszköz teljesítményének mérése, arról külön, a ***8.*** fejezetben olvashatunk.

A transzformáció kimenete a korábbiaknak megfelelően nem lesz hibamentes. Azonban a fordítási hibák jelentős része apró tévedésekből adódik össze, amelyeket kézzel könnyű megszűntetni. Ilyen például egy helytelenül eltalált változó típus, amely minden műveletnél inkompatibilis típust fog jelezni.

Az egyes szoftvereknél meg van adva a kódsorok száma is. Mivel a transzformáció a kommenteket is érinti, ezért a megadott értékek a kommentek és a tényleges kód sorainak összegét jelenti. Ezt minden esetben az ingyenesen letölthető *Cloc*[17]*(Count lines of code)* programmal határoztam meg.

A generált projektek java forráskódja elérhető a következő linken:

https://github.com/Norberts11/MoodernizeExamples

7.1 C-buffered-tree

Az első projekt célja egy *Buffered Tree*[18] nevezetű adatstruktúra implementálása C nyelven. A projekt összesen 2557 releváns sort tartalmaz. ***A C-buffered-tree*** elérhető a következő linken:

https://github.com/yuyuyu101/C-Buffered-tree

A transpiler által generált osztályokat szemlélteti a ***10. ábra.***

[17] https://github.com/AlDanial/cloc

[18] http://www.cs.cmu.edu/~guyb/realworld/slidesF10/buffertree.pdf

10. ábra: A C-buffered-tree generált osztályai.

A transpiler által generált fordítási hibák ránézésre soknak tűnhetnek, de ezek alkalmanként csoportosan, 1-1 hiba miatt következnek be. Ha pld. egy pointer típust a transpiler rosszul talált el és tömbként transzformálta egyszerű változó helyett, akkor minden egyes előfordulásnál kapunk egy errort, miközben a megoldás egyetlen sor módosítása. Ezek kézzel is könnyen elvégezhető helyesbítések.

Sok hibát generálnak továbbá a beépített adattípusok (pl. size_t) és függvényhívások is. A hívások esetén az ***5.3.*** fejezetben ismertetett módon könnyíthető meg a hívások lecserélése a Java-beli megfelelőjükre.

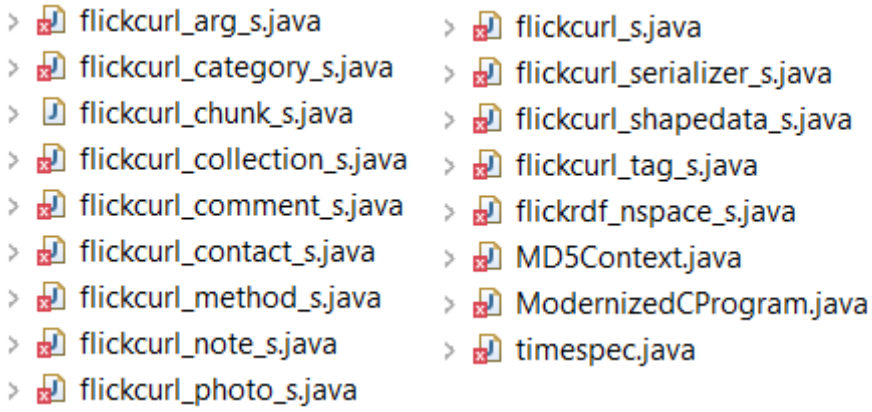
7.2 Flickurl

Ez a szoftver egy megoldást nyújt a *Flickr API*[19] kezeléséhez C nyelven. Ez egy hálózati kommunikációs könyvtár, amely kezeli a kéréseket, tokeneket és meghívja az API megfelelő műveletét. A projekt összesen 33669 releváns sort tartalmaz. A ***Flickurl*** elérhető a következő linken:

https://github.com/dajobe/flickcurl.

A transpiler által generált osztályokat szemlélteti a ***11. ábra.***

[19] https://www.flickr.com/services/api/



11. ábra: A Flickurl generált osztályai.

Az előző projekttel ellentétben a ***Flickurl*** már használ külső könyvtárakat is (***libcurl*** és ***libxml2***), melyek nincsenek csatolva a projekthez. Bár azok forráskódja beszerezhető lenne más forrásból, a transzformációt most azok nélkül futtattam le azért, hogy lássuk, milyen gondokat tud ez okozni. A kód viszonylag sok olyan hibát tartalmaz, ami erre vezethető vissza. Ilyenek pl. a fel nem ismert típusok (Object-ek), illetve „furcsa" üres helyek a kódban. Ezek a korábbiakban említett üres kifejezések, melyek a hiba elszigetelését szolgálják. Ilyen például a következő kódsor is, amely egy nem elérhető típusnév miatt üres „*new*" kifejezést tartalmaz.

```
nodes = new ();
```

Ezek azonban szemléletes és egyértelmű hibák, amelyek kézzel könnyen kijavíthatók.

7.3 Vim

A *Vim* a jól ismert linux alapú szövegszerkesztő. Ez jóval nagyobb szabású projekt mint az előzőek, összesen 419493 sornyi C kódból és kommentből áll.

Ebből a transpiler 240 osztályt generál, melyek összesen 253459 sort tartalmaznak. A sorok számának csökkenésének jelentős részét a preprocesszor utasítások elhagyása okozza.

Nézzük most meg ennek a projektnek a kódját kicsit részletesebben, különös tekintettel a ***6.4*** fejezetekben leírt konverziós kérdésekre.

A projekt jópár *data class-t* tartalmaz, de van sok olyan is, ahova sikerült metódusokat szervezni a paraméterlista és visszatérési érték vizsgálata alapján. Ilyen pl. a *taggy.java*

következő metódusa. Ez, mivel void visszatérési értékű, a paraméterlista alapján került az osztályba.

```
public void tagstack_clear_entry() {
     Object generatedTagname = this.getTagname();
     do {
          if ((generatedTagname) != ((Object) 0)) {
               ModernizedCProgram.vim_free(generatedTagname);
               (generatedTagname) = ((Object) 0);
          }
     } while (0);
          Object generatedUser_data = this.getUser_data();
          do {
               if ((generatedUser_data) != ((Object) 0)) {
                    ModernizedCProgram.vim_free(generatedUser_data);
                    (generatedUser_data) = ((Object) 0);
               }
          } while (0);
     }
```

Ezen a kódon látszik két hiányosság is: az említett NULL makró feloldása és a while logikai feltételének hibája, ahol nem sikerült átalakítani az integer értéket booleanra. Ezen felül létezik néhány üres típus is és olyan is, amely az eredeti típusnevet szerepelteti, de ez a típus nyilvánvalóan nem létezhet a generált kódban (pl. size_t).

A C alapvető típusainak konverzióját jól kezelte a transpiler. A struktúra típusok is többnyire jó helyre kerültek, bár létrejött jónéhány Object típus is. Ezek vagy a ”void *” típus képei, vagy, ami valószínűbb, fel nem ismert típus eredményei (pl. külső függőség struktúrája). A *Vim* tartalmaz a korábban említett hexadecimális és normálalakban megadott számokat is, amelyek “-1024”-es hibakóddal kerültek a kódba.

A pointerek kezelése is összességében jól sikerült, bár vannak rosszul kezelt esetek. Olyan is van, amely tömbként lett kezelve de egyszerű referenciaként kellett volna és fordítva is.

Az elhagyott paramétereken hívott függvények tulajdonosainak lecserélése *this* kulcsszóra szintén sikeresen megtörtént. Erre tartalmaz példát az előző kódrészlet is. A *get* hívásokhoz bevezetett generált referenciák is működnek, erre is van példa az említett kódban. Bár a típust nem sikerült felismerni, a referenciagenerálás és a további referenciák lecserélése rendeltetésszerűen működik.

8 Skálázhatósági vizsgálatok

A szoftvermodernizáció területének fontos kérdése, hogy az eszközök hogyan teljesítenek nagyméretű, akár több százezer sorból álló alkalmazásokkal szemben. Alapvetően két mérőszám fontos, amit vizsgálnunk kell: a transzformáció futásidejét és memóriahasználatát.

Ennek érdekében skálázhatósági vizsgálatokat végeztem valós, komplex, nyílt forráskódú C nyelvű kódbázisokon. A vizsgálat során összesen 6 projekt transzformációján végeztem méréseket az ingyenesen letölthető ***visualvm***[20] eszközzel, amely JVM processek különböző adatait képes monitorozni. A projektek releváns adatait a következő táblázat szemlélteti.

5. táblázat: A vizsgált projektek adatai.

Név	Elérhetőség	Sorok száma	Transzformált sorok száma
Rufus	***https://github.com/pbatard/rufus***	***86199***	***75182***
Hashcat	***https://github.com/hashcat/hashcat***	***175696***	***167300***
Curl	***https://github.com/curl/curl***	***178614***	***104038***
Git	**https://github.com/git/git**	***229125***	***234898***
Obs-studio	**https://github.com/obsproject/obs-studio**	***240076***	***203472***
Vim	**https://github.com/vim/vim**	***419493***	***253459***

Észrevehetjük, hogy a transzformált sorok száma nem minden esetben ugyanúgy változik az eredeti sorok számához képest. Ez több tényezőtől is függhet. Az *include* utasítások elhagyása csökkenti a sorok számát. Ha egy alkalmazásban sok a nem kezelt elem (preprocesszor utasítások), akkor lényegesen kevesebb sor lehet a transzformált projektben. Ha nagyon sok az adatelérés, akkor a generált getter utasítások miatt pedig nőhet a sorok száma. Ez az arány tehát nagyban függ a vizsgált alkalmazás szerkezetétől és logikájától.

A projektek transzformált verziói elérhetőek a következő github linken:

https://github.com/Norberts11/MoodernizeExamples

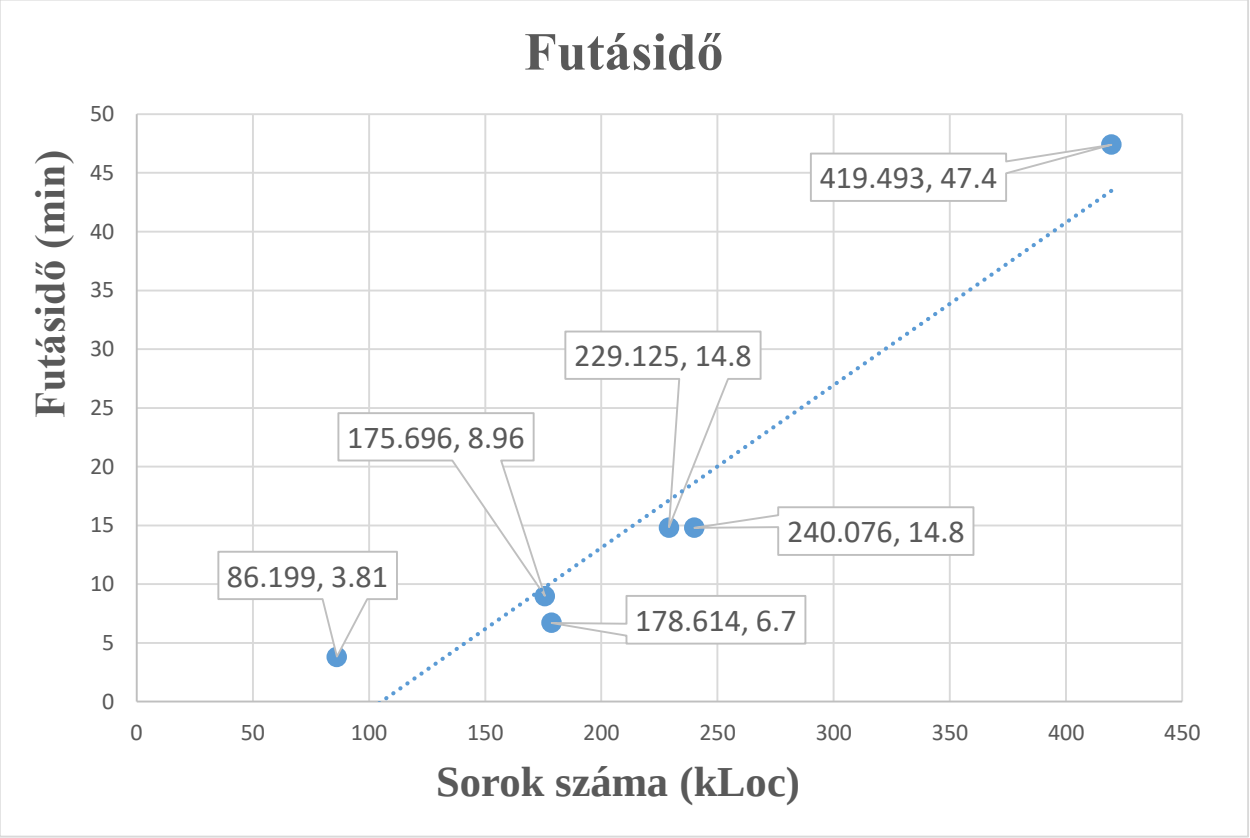
[20] https://visualvm.github.io/

A vizsgálatokat a következő releváns hardveradatok mellett végeztem el:

- Inter(R) Core(TM) i7-7700HQ CPU @ 2,8GHz 2.81 GHz
- 16GB RAM

A mérések eredményeit diagrammokon szemléltetem. Természetesen az eredményeket akármilyen folytonos vonallal összekötni hiba lenne, hiszen az egyértelmű trendet feltételezne az adatok alakulásában. A szemléltetés kedvéért azonban mindkét diagrammon elhelyeztem egy lineáris trendvonalat. Ez azt mutatja meg, hogyan alakulnának a mért értékek, ha a meglévő értékek alapján tökéletes linearitást feltételeznénk.

A futásidőre vonatkozó vizsgálatokat a ***12. ábra*** mutatja.



12. ábra: A transzformáció futásideje.

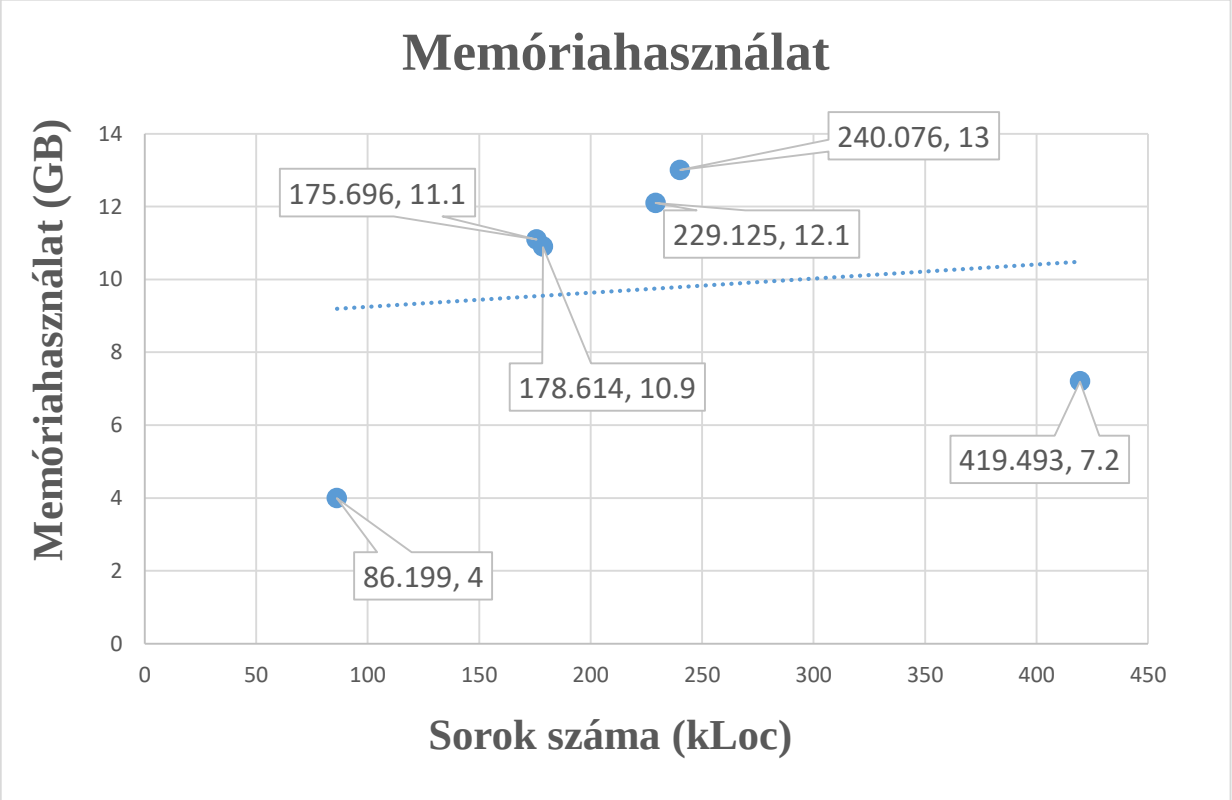
A vízszintes tengelyen a sorok számát láthatjuk 1000-es egységben mérve, a függőlegesen a transzformáció futásidejét percekben megadva. Az adatok feliratában az első érték a vizsgált szoftver sorainak számát jelenti, a második a transzformáció

futásidejét percben mérve. Azt láthatjuk, hogy a minimális mért érték 86199 sornál 381 perc volt, a maximális 419493 sornál 47.4 perc.

Az adatok trendjét elemezve látható, hogy nem teljesen lineáris, mert nem illeszkednek a megadott lineáris trendvonalra. Azt is észrevehetjük azonban, hogy minden mért adat kb. hasonló eltérést mutat a lineáris tendenciától és ez az eltérés nem jelentős mértékű. Emiatt állíthatjuk, hogy a transzformáció futásideje ***lineárisan közelíthető*** tendenciát mutat.

A futásidőre vonatkozó mérés eredményeit összegezve a transpiler jól teljesített a vizsgált szoftvereken. Egy 420.000 sorból álló szoftver esetén 47 perces futásidő jó eredménynek számít. A lineárisan való közelíthetőség pedig azt jelzi, hogy a még nagyobb alkalmazásokkal szemben is nagyjából hasonló tendenciát mutatna a transzformáció futásideje.

A memóriahasználatra vonatkozó méréseket a ***13. ábra*** szemlélteti.



13. ábra: A transzformáció memóriahasználata.

A vízszintes tengelyen szintén a sorok számát láthatjuk 1000-es egységben megadva, a függőlegesen pedig a transzformáció maximális memóriafogyasztását GB-ban mérve. A minimális mért érték 86199 sornál 4GB volt, a maximális 240076 sornál 13GB. Itt észre

kell vennünk, hogy a futásidővel ellentétben a legnagyobb memóriafogyasztása nem a legtöbb sorból álló szoftvernek volt. Sőt, annak volt a 2. legkevesebb. Ebből következik, hogy a tendencia messze nem lineáris függvénye a sorok számának. Ráadásul nem is monoton növekvő függvénye, hiszen kevesebb sorból álló alkalmazások több memóriát képesek fogyasztani, mint a nagyobb társaik.

Ennek az oka az, hogy a memóriafogyasztás nem csak a sorok számától függ. Fontos az alkalmazások felépítése és a használt logika komplexitása is. Ez bizonyos mértékben igaz a futásidőre is, de messze nem annyira, mint a memóriahasználatra.

A memóriafogyasztás szempontjából fontos tényező, hogy hány fájlból áll az alkalmazás. Ugyanis egy sok fájlból álló szoftver sokkal több memóriát képes fogyasztani még akkor is, ha összesen kevesebb sort tartalmaz. Ennek az oka az AST-k elkészítésében keresendő. Több fájl több fát jelent, melyek több memóriát foglalnak, mintha egy fában lenne több elem. Ennek szemléltetésére tekintsük a következő táblázatot.

6. táblázat: A vizsgált projektek további adatai.

Név	Forrás- és fejlécfájlok száma	Sorok száma	AST-k memóriaigénye (GB)	Maximális memóriahasználat (GB)
Rufus	***290***	***86199***	***3.4***	***4***
Hashcat	***676***	***175696***	***10.4***	***11.1***
Curl	***691***	***178614***	***9.1***	***10.9***
Git	***657***	***229125***	***10.8***	***12.1***
Obs-studio	***1014***	***240076***	***11.8***	***13***
Vim	***243***	***419493***	***6.2***	***7.2***

Az AST-k memóriaigényeit az elemzés (parsing) végének pillanatában mért memóriafogyasztás aktuális értéke adja.

Ezekből az adatokból már jól látszik a fájlok száma és a memóriafogyasztás közötti összefüggés. A kb. azonos mennyiségű fájlokat tartalmazó projektek kb. hasonló tendenciát mutatnak memóriahasználatban. Így már érthető az is, hogy a legnagyobb szoftver transzformációja miért foglalt lényegesen kevesebb memóriát a nála kisebb szoftvereknél.

Ebből a táblázatból azonban más következtetés is levonható. Vegyük észre azt is, hogy az AST-k kialakításához szükséges memória az abszolút minimum, amivel a transzformáció elvégezhető. A fáknak el kell férniük a memóriában, ezen nem tudunk

spórolni. Ennek fényében azt is láthatjuk, hogy a maximális fogyasztás túlnyomó részét az AST-k kialakítása és memóriában tartása adja. Ezért a transzformáció memóriafogyasztása (amely a két érték különbsége) nem növelte meg jelentősen az egész folyamat memóriahasználatát. A legkisebb esetben a transzformáció 0.6GB (*Rufus*), a legnagyobb esetben pedig 1.8GB (Curl) memóriát fogyasztott. A transzformáció memóriafogyasztásának tehát már nem a fájlok száma a fő befolyásoló tényezője, hanem a logika komplexitása.

Összegezve a mérés eredményeit azt mondhatjuk, hogy a transzformátor memóriahasználat szempontjából is jól teljesített. Ezt leginkább az jelzi, hogy a minimálisan szükséges memóriahasználatot (AST-k felépítése) nem növelte meg jelentősen a transzformáció hátralévő részének futása.

9 Összefoglalás

Dolgozatomban bemutattam egy módszert, amely procedurális felépítésű kódot képes objektumorientált szerkezetű kódra átalakítani. A módszert modell-vezérelt szemlélettel közelítettem meg. Ez azt jelenti, hogy az eredeti alkalmazásból egy olyan modellt készítettem, amely egy ugyanolyan szemantikával rendelkező, objektumorientált szemléletű alkalmazást ír le. Az átalakítás során a forrásnyelv C, a célnyelv Java volt.

A transzformáció egyik legfontosabb kérdése az objektumorientált paradigmaváltás volt. Ez azért volt fontos, hogy minél áttekinthetőbb, karbantarthatóbb és bővíthetőbb kódot kapjunk. Ezt a felépített modellen alkalmazott modell-transzformációk sorozatával értem el.

Az alkalmazott transzformációk a C nyelv struktúráit osztályoknak feleltetik meg, valamint a globális függvényeket próbálják meg ezekhez az osztályokhoz rendelni, mivel általában a procedurális felépítés miatt az adatstruktúrákon végzett műveletek különálló függvényekbe vannak szervezve. Ez a paraméter-és visszatérési érték vizsgálat célja.

Azokat a függvényeket, amelyeknél a hozzárendelés nem sikerült, további vizsgálatnak vetem alá, hogy úgy szervezzem őket egy külön osztályba, hogy az alkalmazás minél kevesebb komponense függjön ettől az osztálytól. Ezt a függvények közti hívási láncok elemzésével vizsgálom.

Bemutattam egy módszert, amely segítségével a procedurális kód könyvtári hívásait le tudjuk képezni a célnyelv megfelelő hívásaira. Erre egy szakterületi nyelvet készítettem ***Xtext*** segítségével. Az ezen a nyelven írt szkriptekből transzformációs modell generálható, majd annak felhasználásával az AST-k bejárása során a könyvtári hívások lecserélhetők.

Dolgozatomban bemutattam egy általam készített transpilert is, amely a kidolgozott módszertant valósítja meg a gyakorlatban. A transpiler az ismertetett transzformációk végrehajtásával alakítja ki a C nyelvű program Java nyelvű, objektumorientált szerkezetét. A transpiler elkészült, és a ***6.5*** fejezetben leírt hiányosságoktól eltekintve a nyelvi elemek többségét képes helyesen kezelni, ezért alkalmas valós szoftverek vizsgálatára.

Az eszköz működését szemléltettem és elemeztem valós, nyílt-forráskódú alkalmazások transzformációjának esettanulmányain keresztül. Végezetül komplex, nagyméretű szoftverek átalakításán keresztül méréseket végeztem a transpiler teljesítményének megállapítására. A mérések eredményei alapján a transpiler a futásidő és a memóriahasználat szempontjából is jól teljesít valós, komplex projektekkel szemben.

9.1 Az eredmények értékelése

A módszerem modell-vezérelt megközelítésének nagy előnye, hogy számos gazdag funkcionalitást biztosító modell-vezérelt eszközt használhatunk a transzformáció során, amelyek hatékonyabbá tehetik a transpiler fejlesztését. Ilyenek voltak pl. a dolgozatban bemutatott ***Eclipse Modeling Framework (EMF)*** és ***Xtext*** keretrendszerek, de természetesen a továbbiakban más eszközöket is lehet használni. Ezen felül a modell-vezérelt megközelítésnek köszönhetően a transzformáció során egy nyelvfüggetlen objektumorientált modellt tudunk építeni. Így a transzformáció jelentős része nyelvfüggetlen módon tud működni. Ez nagyban megkönnyíti a transpiler különböző forrás- és célnyelvekre való kifejlesztését.

Az alkalmazott Eclipse környezethez könnyű bővíthetősége miatt számos modell-vezérelt eszköz érhető el, ezért ez ideális fejlesztőkörnyezet a problémakör modell-vezérelt vizsgálatához.

9.2 Továbbfejlesztési lehetőségek

A bemutatott megoldást a jövőben számos módon ki lehet bővíteni. A módszertanban végezhetünk további vizsgálatokat, amelyeket a transpiler működése során vizsgálhatunk és szemléltethetünk. Ezen felül a hiányzó vagy helytelenül kezelt nyelvi elemeket is meg kell valósítani.

Ezek a lehetőségek pl. a következőkre terjedhetnek ki:

- A ***6.5.*** fejezetben ismertetett hiányosságokat és limitációkat ki kell küszöbölni. Ez egyben a még nem támogatott elemeket és a helytelenül kezelt eseteket is jelenti.
- A transpilert további modell-vezérelt vagy kódelemző eszközökkel integrálhatnánk. Érdekes lehet pl. a ***3.3.3*** fejezetben látott *CodeCompass*-al vagy

hasonló eszközzel való integráció. Egy kódelemző eszköz által összegyűjtött extra információk hasznosak lehetnek a transzformáció során.

- A módszertanban az ***5.2.3*** fejezetben bemutatott hívási láncok vizsgálatát tovább gondolhatnánk a ***3.2.1*** fejezetben látott klaszterező megoldások használatával.
- A módszertan jelenleg minden generált fájlt egyetlen *packagebe* szervez. A generált osztályok közötti függőségek felderítésével kialakíthatnánk automatikusan egy optimális felosztást, amely minimalizálja az egy *packagebe* került osztályok közötti függőségeket.
- Az ***5.3*** fejezetben leírt szakterületi nyelvet ki lehetne bővíteni, hogy bonyolultabb eseteket is le lehessen írni vele.

Irodalomjegyzék

[1] Ecore. Forrás: https://wiki.eclipse.org/Ecore.

[2] B. Iván, *Formális nyelvek*, Budapest: Typotex, 2002.

[3] A. Aho, R. Sethi, J. Ullman és M. S. Lam, *Compilers: Principles, Techniques, and Tools*, Pearson Education Inc, 2006.

[4] N. Wirth, *What can we do about the unnecessary diversity of notation for syntactic definitions?*, Federal Institute of Technology (ETH), Zürich: ACM, 1977, pp. 822-823.

[5] A. A. Terekhov és C. Verhoef, *The Realities of Language Conversions*, Los Alamitos, California: IEEE Computer Society Press, 2000, pp. 111-124.

[6] M. van den Brand, A. Sellink és C. Verhoef, *Current parsing techniques in software renovation considered harmful*, Ischia, Italy: IEEE Computer Society, 1998, p. 108.

[7] B. S. Everitt, S. Landau és M. Leese, *Cluster Analysis*, Wiley Publishing, 2009.

[8] Y. Zou és K. Kontogiannis, *A Framework for Migrating Procedural Code to Object-Oriented Platforms*, Macao, China: IEEE, 2001, p. 390.

[9] I. G. Czibula és G. Czibula, *Unsupervised transformation of procedural programs to object-oriented design*, Acta Universitatis Apulensis, 2011.

[10] A. van Deursen és T. Kuipers, *Identifying objects using cluster and concept analysis*, Los Angeles, California: ACM, 1999, pp. 246-255.

[11] G. Snelting, *Concept Analysis - a New Framework for Program Understanding, SIGPLAN Not.*, pp. 1-10, 1998.

[12] H. Sneed, *Migrating PL/I Code to Java*, Washington, DC: IEEE Computer Society, 2011, pp. 287-296.

[13] P. Newcomb és G. Kotik, *Reengineering Procedural Into Object-Oriented Systems*, Toronto, Canada: IEEE Computer Society, 1995, p. 237.

[14] H. Gall és R. Klösch, *Program Transformation to enhance the Reuse Potential of Procedural Software*, Phoenix, Arizona: ACM, 1994, pp. 99-104.

[15] *mtSystems*. Forrás: https://www.mtsystems.com/.

[16] *Tangible Software Solutions*. Forrás: https://www.tangiblesoftwaresolutions.com/

[17] *Ericsson Codecompass*. Forrás: https://github.com/Ericsson/CodeCompass.

[18] Z. Porkoláb és T. Brunner, *The Codecompass Comprehension Framework*, Gothenburg, Sweden: ACM, 2018, pp. 393-396.

[19] *Ericsson Codechecker*. Forrás: https://github.com/Ericsson/codechecker.

[20] L. Vogel, *Eclipse Rich Client Platform*, Lars Vogel , 2015.

[21] R. Hall, K. Pauls, S. McCulloch és D. Savage, *OSGi in Action: Creating Modular Applications in Java*, Manning Publications, 2011.

[22] P. Harjivanbhai Ghumalia és M. Purnank Ghumalia, *C/C++ Software Development with Eclipse*, Purnank & Meera, 2014.

[23] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman és P. McCarthy, *The Java Developer's Guide to Eclipse, 2nd Edition*, Addison-Wesley Professional, 2004.

[24] D. Steinberg, F. Budinsky, M. Paternostro és E. Merks, *Eclipse Modeling Framework, 2nd Edition*, Addison-Wesley Professional, 2008.

[25] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, Birmingham: PACKT Publishing, 2013.

[26] M. Fowler, *Domain-Specific Languages*, Addison-Wesley Professional, 2010.

[27] T. Parr, *The Definitive ANTLR 4 Reference*, Pragmatic Bookshelf, 2007.

[28] L. Dykes és E. Tittel, *XML For Dummies*, For Dummies, 2005.

[29] *Xtend*. Forrás: https://www.eclipse.org/xtend/documentation/index.html.

[30] F. Appel, *Testing with Junit*, Packt Publishing, 2015.

[31] *Scoping*. Forrás: https://users.cs.cf.ac.uk/Dave.Marshall/PERL/node52.html