FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.

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

Horváth István Máté

KÓDGENERÁTOR KÉSZÍTÉSE ECLIPSE PLATFORM HASZNÁLATÁVAL

Kisfeszültségű folyamatirányítók

KONZULENS

Kövesdán Gábor

BUDAPEST, 2018

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Horváth István Máté**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2018. 11. 26.

…….
Horváth István Máté

Összefoglaló

A szoftverfejlesztők az informatika kezdete óta próbálnak olyan technikákat alkalmazni, melyeknek segítségével egyszerűbbé és gyorsabbá válhat egy szoftver létrehozásának a folyamata. Ezeknek a módszereknek az egyik modern változata a modell alapú fejlesztés.

A fejlesztők az elkészítendő szoftverhez modelleket készítenek a kívánt részletességgel, és e modelleket felhasználják a fejlesztés minden fázisában, főként a tervezés és az implementálás fázisában. A szoftver modelljének a megalkotásában az informatikában nem jártas egyének is részt vehetnek, és sok egyéb előnye is van egy modell használatának.

Szakdolgozatom fő témája a modellalapú szoftverfejlesztés megismerése és a bemutatása, mint modern fejlesztési technika Ehhez egy régóta előforduló, de többnyire megoldott problémát újból megvizsgálok, mely ismételten meg lesz oldva, de most modellek segítségével.

Ez a probléma nem más, mint a karakterkódolások és a karakterkészletek sokszínűségének bonyodalma. Rengeteg megoldás és szoftver készült, de egyik sem modell alapokon. Miután megvalósítom a saját megoldásomat a karakterkódolások problémájára, lesz egy eredményem, mennyivel egyszerűbb és átláthatóbb ez az újfajta fejlesztési módszer.

Abstract

Ever since the beginning of Information Technology, software developers try to come up with new technics, that will help the process of the development for a software much faster and much simpler. One of these technics' modern variant is the model-driven engineering.

Developers make models for their software to be prepared with as many details as they want. Later, they use these models in every phase of the development, especially in the designing and the implementation phase. People, who are not familiar with Informatics, can help creating the software's model, and there are a lot other advantage in the usage of a model.

The main article of my thesis is to study and present model-driven engineering, as a modern development technic. For this purpose, I will expose an old problem and I will re-solve it once again but with the help of models.

This problem is nothing less, than the problem of character encodings, character sets and their diversity. A lot of solutions and softwares have been already made, but none of those did actually use models. After I have created my own solution for the problem of character encoding, I will acquire an outcome of how easy and clean this new style of development can be.

1 Bevezetés

Egy szoftver fejlesztése során egyáltalán nem újszerű ötlet diagramok készítése. Például, a szoftvertervezés fázisában régóta használunk UML diagramokat, hogy az objektumorientált programozási elemekről tervezési döntéseket nagy hatásfokkal jegyezhessünk fel. Továbbá, a szoftverhez kapcsolódó dokumentációkban is több szerepet kapnak, mint egy szöveges leírás. Azonban ezek a diagramok nem használhatóak fel olyan mértékben, mint egy modell. Egy modell sokkal több funkciót képes ellátni, és sokkal több lehetőség van benne.

A modellalapú fejlesztés egyik fajtája a Domain Specific Language alapú alkalmazások, röviden DSL-ek. Egy DSL külsőre egy programozási nyelv, azonban nem feltétlenül rendelkezik saját fordítóval és nem is annyira sokoldalú, mint egy megszokott programozási nyelv. Egy DSL legtöbbször egy másik programozási nyelvre fordul át, vagy legalábbis egy adathalmazra, amely egy másik nyelv felhasználhat. A DSL-ek tipikusan egy-egy fajta, konkrét feladat betöltésére lettek megalkotva.

A DSL-ek létjogosultsága abban rejlik, hogy a bennük való kódírás sokkal egyszerűbb és letisztultabb lehet egy átlagos nyelvhez képest, így laikusok is képesek benne programozni. Bevonhatja a szoftver megrendelőjét, a steakholdereket a fejlesztés folyamatába, megkönnyíti a fejlesztőkkel való munkát, azáltal, hogy az ügyfél konkrétan tudja közölni az igényeit az alkalmazás doménjén keresztül.

DSL lehet például egy építészeti egységeket, kémiai vagy gazdasági folyamatokat leíró nyelv, egy programozást oktató vagy statisztikákat készítő nyelv. De a weboldalak HTML nyelve és a Linuxban használatos szkriptkészítés is DSL-nek tekinthetők, bár ezek jóval általánosabbak, mint amit DSL-nek szokás nevezni. A DSL másik használatos, informális neve a mini-language.

Az informatika gyorsabb ütemben fejlődik, mint a többi tudományág, aminek megvannak a nem kívánt hatásai is. Egy a sok közül, mellyel a szakdolgozatom is foglalkozik, a karakterkódolások sokféleségének a problémája.

Minden egyes szöveg vagy adat, amit az informatika világában létrehozunk, számok formájában kerül tárolásra, és egy karakterkódolás határozza meg, hogy melyik betűhöz vagy jelöléshez, melyik szám tartozik. Az informatika kialakulásának

legelejétől a mai napig rengetegféle kódolási szabványok jöttek létre és a meglévők bővülnek. Kezdetben igények szerint, földrajzi elhelyezkedés, kulturális háttér alapján jöttek létre, például az ASCII, amely kizárólag az angol billentyűzeten található karaktereket foglalja magába, vagy a JIS, mely a Japánban használt írásjeleket kódolja.

Napjainkban már inkább az egységes, szabványos és mindenhol használandó kódolások kezdenek elterjedni. Az Unicode kódolás áll e tekintetben a legjobb helyen, ez a legismertebb és a legrészletesebb kódolás, mivel a legkézenfekvőbb karakterekkel kezdve, az ázsiai írásjeleken át, az egyiptomi hieroglifákon keresztül, az emojikkal bezárólag minden megtalálható már benne.

A karakterkódolással kapcsolatos egyik problémát az elavult karakterkészletek jelentik. A régebbi kódolások nem tudnak egyik napról a másikra eltűnni. Nem használt állományok még mindig ilyen kódolásokban vannak tárolva, melyek használhatatlanná válhatnak. Egész adatbázisok lehetnek elérhetetlenek, más kontinensen lévő nemzetek régi fájljaihoz nem lehet hozzáférni, könyvtárban tárolt szöveges állományokat nem tudunk elolvasni, stb… Pusztán azért, mert olyan régi kódolást használnak, amelyeket gépi segítség nélkül nem lehet ismerni.

A másik problémát hasonló alapon az Internet jelenti. A különböző kódolások miatt nem egyszerű e-maileket írni, weblapokat letölteni, alkalmazásokat futtatni. Mongol barátomnak angolt, vagy angol karakterekre interpretált mongolt kell használni, hogy a családjának üzeneteket küldhessen, pusztán azért, mert a webes szolgáltatásokon és alkalmazásokon nincsenek támogatva a cirill betűk.

Összefoglalva, a karakterkódolások közti átvitel problémát jelent, és nincs rá megfelelő eszköz, amely ezt meggyorsíthatná.

A karakterkódolás konvertálóhoz könnyen elképzelhető valamilyen DSL. Egy átkódolás lényegi része, hogy milyen bitsorozatból milyen másik bitsorozat lesz. Ezek leírása egyszerűen és zajmentesen leírhatóak lehetnének egy DSL-ben. Szakdolgozatomban egy ilyet szeretnék megvalósítani. Az én DSL nyelvemben megadható lehetne a kiinduló kódolás, a célkódolás és a konvertálandó bitsorozatok szimpla szöveges formátumban, bármilyen zavaró szintaxis nélkül.

Szakdolgozatom témája egy kódgenerátor készítése. A fent említett DSL-t már lehetne arra használni, hogy a megírt átkódolásból valamilyen közismert programozási nyelven megírt API-t (application programming interface) generáljak, hozzáadva azon „zajokat”, amely a DSL eltávolított. A programot nem kézzel kellene megírni, hanem generálnám, így a hosszan megírt karakterátkódolási szabályokat könnyen lehetne kezelni.

A feladatot egy DSL szerkesztőben kell elkészíteni, mely az Eclipse platformon elérhető. A szerkesztőnek, vagyis Eclipse pluginnek a neve Xtext, melyben gyorsan, egyszerűen és mégis eredményesen lehet DSL-t készíteni. Az Xtext nagymértékben támaszkodik az Eclipse Modelling Frameworkre, mely az Eclipse modellezéshez kapcsolódó legelterjedtebb pluginje. Az Xtext segítségével Java nyelvben lehet megírni a DSL részeit és rengeteg segédosztályt és segédfüggvényt biztosít, hogy a DSL miden funkcióját az igényekhez igazítsuk.

A konkrét igénnyel Kövesdán Gábor konzulensem állt elő. Felhívta a figyelmemet, hogy a beágyazott rendszerekben is előforduló FreeBSD-ben az elérhető, karakterkódolásokhoz használt konvertálók a mai technikához képest nem számítanak praktikus megoldásoknak. Ezeket, a már megoldott problémákat, modern eszközökkel egyszerűbb és letisztultabb formában lehetne elkészíteni. Egy C nyelvben megírt függvénycsoportra lenne szükség, melyekben lekérhetjük az egyes átkódolásokban használatos bitsorozatokat és a konvertáláshoz szükséges egyéb információkat. A függvények számára az adathalmazt az elkészített DSL és kódgenerátor használatával tennénk elérhetővé. Ezt elég érdekesnek találtam ahhoz, hogy szakdolgozatom témájának válasszam, és remélhetőleg egy open source projekthez is hozzájárulhatok.

2 Elméleti háttér

2.1 Karakterkódolás

Karakterkódolásnak nevezzük azt, amikor betűk, számok és karakterek egy halmazához valamilyen kódot rendelünk. Ez a kód kontextustól függően lehetnek például ábrák (a szabadkőműves ábécé) vagy fizikai impulzusok (Morse-kód). Az informatikában természetes számokat használunk, mivel a számítógépek ezeket tudják a legegyszerűbben tárolni. Egy karakterkódolásnál használatos struktúrákat a legelterjedtebb karakterkódoláson, a Unicode szabványon keresztül szeretném ismertetni.[1] Ez a modell egy másik modellt, az IAB modellt egészíti ki.[2]

Az első kérdés, ami egy karakterkódolásnál felmerül, hogy melyik karaktereket szeretnénk lekódolni. Ezen karakterek halmazának a neve Abstract Character Repertoire (ACR). Bár a definíció nem követeli meg, mégis elterjedt, hogy a karakterek halmaza egyazon ábécé betűiből és az ábécéhez tartozó írásjelekből áll. Egy ACR lehet zárt vagy nyitott. A zárt repertoár megváltoztatására vagy kibővítésére nincs lehetőség, míg a nyitott repertoárhoz hozzá lehet adni újabb karaktereket. Az IAB modellben az ACR a karakterkészletnek (character set) felel meg.

Következő szint a Coded Character Set (CSS). A CSS egy leképzés (mapping), ahol az absztrakt karakterekhez nemnegatív egész számokat rendelünk. A számoknak nem kötelező egymást követőeknek lenniük. A karakterek és a számok között 1-1 kapcsolat lehet. Ha egy karakterhez hozzárendelünk egy számot, akkor az a karakter része lesz a kódolt karakterkészletnek (coded character set). Ekkor a karaktert kódolt karakternek (coded character), a számot pedig kódpontnak (code point) nevezhetjük. (ISO) A kódolt karakterkészletet másik elnevezése a karakterkódolás (character encoding), vagy az IAB modellből ismert kódlap (code page).

A Character Encoding Form (CEF) szintén egy leképezés, mely a CSS-ben használt kódpontokat rendeli kódegységek sorozatához (sequences of code unit). Egy kódegység egy egész számot jelent, mely egy meghatározott bináris szélességet foglal el egy számítógépes architektúrában, például egy 8-bites bájt. Vagyis a CEF azt határozza meg, hogy az egyes karakterek hány kódegységen lehetnek tárolva, és ez a karakterkódolás egyik legfontosabb aspektusa. Kódegységek sorozatának mérete

alapján megkülönböztetünk fix szélességű és változó szélességű CEF-eket. Szemléltetésként, az UTF-8-at említeném meg, ahol a karakterek 1-4 bájton vannak reprezentálva.

Az utolsó szinten a Character Encoding Scheme (CES) található. A CES egy olyan reverzibilis hozzárendelés, melyben a kódegységekhez rendelünk bájtok sorozatát. A CES esetében már foglalkozni kell a számítógépek architektúrájával is, hogy biztosítani tudjuk cross-platfrom esetén is az adatok perzisztenciáját, például, hogy egy architektúra big-endian (UTF-16BE) vagy little-endian (UTF-16LE).

Egy önálló szintet is fel szoktak sorolni, amely a Unicode modelljén kívül helyezkedik el. A Character Maps (CM) a fentebb ismertetett szinteket fűzi össze egy transzformációvá. A CM absztrakt karaktereket alakít bájtok sorozatává, vagyis a CSS, CEF és CES műveleteket végzi el.

Az ACR szintjén lévő karakterek absztrakt karakterek, melyek hagyományok segítségével kialakult fogalmak. Az írásjelek (glyph) az absztrakt karakterekhez tartozó konkrét képek vagy alakzatok. Egyazon absztrakt karakterhez hasonló, de részletekben eltérő írásjelek is tartozhatnak. Ezek a különböző írásjelek hozzák létre a különböző betűtípusokat. Fontos megemlíteni, hogy az írásjelek nem egy-egy kapcsolatban vannak a karakterekkel. Ebből kifolyólag ligatúrák jöhetnek létre, melyek közül a leghíresebb az f-i ligatúra:

fi	fi
Az f és i karakterek ligatúra nélkül	Az f-i ligatúra

2.1. táblázat: Az f-i ligatúra

A karakterkódolások történetét tekintve, az egyik legelső kódolás az American Standard Code for Information Interchange, vagyis ASCII volt. Ez tartalmazta az angol ábécé kis és nagybetűit, számokat, írásjeleket. Azonban más nyelvekben lévő írásjeleket nem, ezért minden ország/nyelv/kultúra elkészítette a sajátját. Japánban több kódolás is elterjedt lett, ezek nem voltak egymással kompatibilisek. Ennek következtében a rosszul dekódolt karakterek semmilyen nyelvre nem hasonlító, olvashatatlan szöveget alkottak.

Ez a jelenség annyira gyakori volt, hogy nevet is kapott: „mojibake”, szó szerint fordítva „karakter transzformáció”.[3]

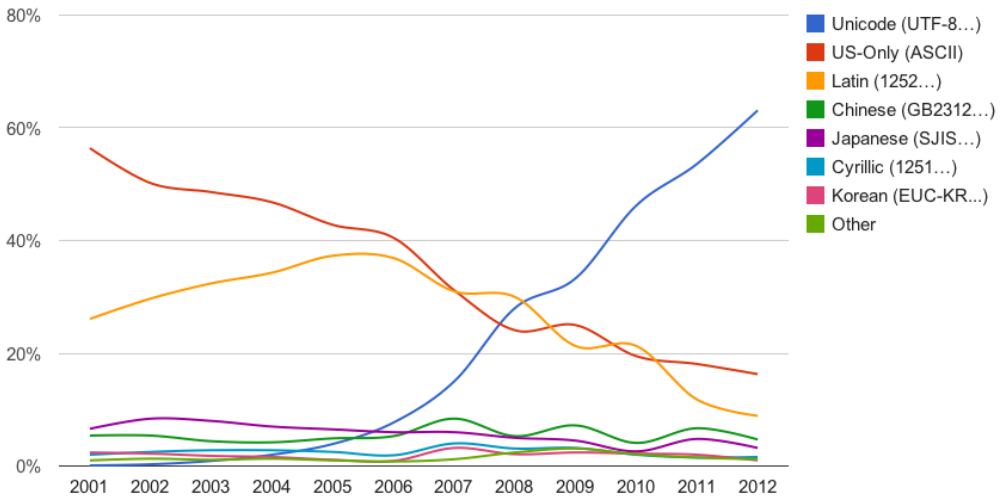
Ez a probléma a World Wide Web megjelenésével még nagyobb formát öltött. Egyértelművé vált, hogy szükséges egy világszerte egységes karakterkódolási szabvány, és ez a Unicode nevet kapta. Az Unicode célja, hogy globálisan elérhető összes karaktert és írásjelet összegyűjtse.

A Unicode több százezer karaktert és kódpontot párosít össze. Ezek tárolásához a kódolásoknak négy bitet kell használniuk karakterenként. Az UTF-32 pontosan így működik. Azonban a legtöbb esetben ez erősen pazarló tárolási módszer, hiszen a leggyakoribb karaktereket egyetlen bájtban is el lehet tárolni. Az UTF-8 valósítja meg ezt az elképzelést.[4]

Karakter	Kódolás	bitek
A	UTF-8	01000001
A	UTF-32	00000000 00000000 00000000 01000001
あ	UTF-8	11100011 10000001 10000010
あ	UTF-32	00000000 00000000 00110000 01000010

2.2. táblázat: Az UTF-8 tömörítése

Az UTF-8 kódegysége változó hosszúságú, így csak annyi bájtot fog használni a karakterkódolására, amennyit az absztrakt karakter kódpontja megkövetel. A kódpontján kívül egyéb biteket is tárolni kell, mellyel azt jelezhetjük a dekódolónak, hogy több bájt tartozik ugyanazon karakterhez. Praktikussága miatt, az UTF-8 a világ egyik legelterjedtebb kódolása lett.[5]



2.1. ábra: Az UTF-8 elterjedése

2.2 Modellezés

A modell definíció szerint, a valóság vagy hipotetikus világ egy részének egyszerűsített képe, amely a rendszert helyettesíti bizonyos megfontolásokkal. Amikor egy modellt képezünk le egy rendszerről, a nem releváns részek elhagyhatóak. Ezzel a rendszer egy kisebb, véges változatát kapjuk, amivel egyszerűbbé és gyorsabbá válik a munka. Például fizikai kísérletben elhanyagolhatjuk a légellenállást. Azonban ezek a részletek torzítják is a modellt a valósághoz képest, és a valós helyzetben nem biztos, hogy helyes működést fog produkálni. Ráadásul, egyes tényezők annyira komplexek, hogy modellezni sem lehet őket, például emberi döntések.

Szoftverfejlesztésnél is van lehetőség modelleket készítésére, ezt modell alapú fejlesztésnek (model-driven engineering) nevezzük. Modell segítségével könnyedén leírható, hogy a szoftvernek milyen komponensei vannak, azok hogyan viselkednek, és milyen kapcsolatban vannak más komponensekkel. Egy modell legfőbb előnye, hogy lehet hozzá nézeteket készíteni, ezeket diagramoknak hívjuk. A diagramok ugyanis megkönnyítik a kommunikációt, amely a fejlesztés egyik legalapvetőbb aspektusa. Diagramok segítségével gyorsan lehet bonyolult tervezési döntéseket feljegyezni, tárolni és átadni két fél között, legyen az ember vagy gép.

A modellek nemcsak informatikában jártas személyek között segítik az együttműködést. Más tudományágak szakemberei értelemszerűen csak a saját szakterületükhöz értenek, és nem jártasak az informatikában vagy a szoftverfejlesztésben. Azonban készíthetőek olyan diagramok, amelyek specifikusan az

adott szakterület jellemzőit használják vagy jelrendszerét hordozzák. Ezáltal más specialistákat, akár stakeholdereket vagy felhasználókat is be lehet vonni a fejlesztésbe.

Egy modell elkészítése nemcsak a tervezés fázisában segíthet. A tervezést megkönnyíti, és átláthatóvá teszi. A modellek egyik tulajdonsága, hogy a fejlesztők szabják meg a modell részletességét. Ha több részlettel gazdagítják a modellt, azt finomításnak nevezzük, ha pedig csökkentik a modell komplexitását, akkor absztrakciót végeznek. A modell mindig igazodni fog ahhoz, hogy a fejlesztők adott pillanatban hol tartanak a fejlesztésben, illetve, hogy mi a tervezés végeztével a kívánt eredmény.

A szoftver implementálása alatt segíthet, ha a diagramokat átolvasva rövid időn belül betekintést kaphatunk a programjaink elvárt működésébe. Futtatható kód generálása is lehetséges, megfelelő feldolgozó és generátor készítésével, és ez a generált programkód meggyorsítja az implementálást.

A modell a tesztelésben, validálásban és verifikálásban is segíthet. Modell készítésével még el nem készült objektumok és komponensek viselkedését lehet szimulálni és méréseket végezni. A modell tesztelhető, hogy minden specifikált követelménynek megfele-e, vagy megvizsgálhatjuk, hogy teszteseteink a modell minden részére
kiterjednek-e. Továbbá, ha a kódgenerátort is helyesen teszteljük, a modell által generált kód is kevesebb hibát tartalmaz.

Végül a dokumentálásnál is előnyt jelent, ha rendelkezünk egy modellel. A modellből készített diagramok önmagukban is rengeteg információt hordoznak, de akár generálhatunk is leírásokat, paramétereket is a szoftverünk számára.

Modellek és diagramok készítésénél meg van szabva, hogy az informatikai fejlesztők vagy más szakemberek milyen elemeket adhatnak hozzá a modellhez. Ezeket az előírások szabványok (pl. UML) korlátozzák. Azt, hogy a modellben lévő egyes komponensek fajtája mi lehet, és a komponensek milyen kapcsolatban állhatnak egymással, a metamodell írja le. A metamodellre tekinthetünk úgy is, mint egy modellnek a modellje.

Metamodellre példa az UML osztálydiagram és objektumdiagram. Az objektumdiagramban lévő objektumok példányosítják az osztálydiagram osztályait és

ugyanahhoz az osztálydiagramhoz több különböző objektumdiagramot készíthetünk. Másik szemszögből vizsgálva, az osztálydiagram leírja, milyen *fajta* elemeket tárolhat egy objektumdiagram, tehát az osztálydiagram az objektumdiagram metamodellje. Hasonló példa egy XML dokumentum, melynek metamodellje az XML séma.

Ezt a gondolatmenetet folytatva, mivel a metamodellek is teljes értékű modellek, ezért azoknak is létezik elkészíthető metamodellje. A metamodellek hierarchiája egy végtelen rekurziót hoz létre, amit egy olyan metamodell oldja fel, amely saját magát írja le, saját maga metamodellje.

A modellezés egyik fajtája a Domain Specific Language-ek (DSL). Egy DSL-re programozási nyelvként lehet tekinteni, azonban kevesebb funkcióval rendelkezik, mint egy átlagos nyelv. A DSL-eket – néhány kivételtől eltekintve – nem általános célokra készítik el, hanem egy konkrét feladat elvégzésére, konkrét terület problémáinak megoldására. Ebből kifolyólag kevés műveletet lehet végezni velük, de eleget ahhoz, hogy az adott feladat teljesíthető legyen. Olyan DSL-ek is léteznek, melyek csupán információk rögzítésére, vagy leírás készítésére szolgál. A DSL-ek legtöbbször nem rendelkeznek saját fordítóval, hanem egy általános célú programnyelvre konvertálódnak át. A DSL-ek másik, kevésbé elterjedt megnevezése a mini-language.

Egy meghatározott feladat elvégzésében sok előnnyel jár a feladathoz készült DSL használata, mint egy megszokott, általános célú programozási nyelv alkalmazása. Ezeket az előnyöket a végrehajtható műveletek korlátozott számának köszönheti a DSL. A DSL-ben történő programozást gyorsabban meg lehet tanulni, mint egy általános célú nyelv használatát. Egy jól kidolgozott DSL átláthatóbb kódot és gyorsabb fejlesztést biztosít. Komplex feladatok elvégzésére, például egy teljes szoftverrendszer megírásához nem létezik DSL, és nem is érdemes ilyen DSL-t készíteni. Azonban többfajta DSL rugalmas használatával bonyolultabb célokat is elérhetünk.

Gyakori, hogy az informatikusoknak olyan szakemberekkel kell együttműködnie, akik nem, vagy keveset értenek informatikához. Az informatikusok pedig az adott szakterületben nem jártasak, ami megnehezíti a fejlesztés számára elengedhetetlen kommunikációt. Ilyen helyzetekre kínálnak megoldást a DSL-ek. Az programozók készíthetnek egy DSL-t, amelyben az idegen szaknyelvet és ismeretlen fogalmakat lehetne megosztani. Az informatikában laikus szakemberek könnyedén

megtanulhatják a DSL használatát, és az informatikusok számára érthető információkat lesznek képesek nyújtani.

Szoftverfejlesztők rengeteg DSL-t használnak munkájuk során, akár tudtukon kívül. DSL-nek tekinthető a HTML nyelv, bash szkriptek, reguláris kifejezések, SQL parancsok vagy az XML dokumentumok, bár ezek sokkal általánosabb célúak. A CSS remek példa a DSL-ek könnyű elsajátítására, mivel vannak olyan weboldalkészítők, akik jártasak a CSS használatában, mégsem tekintik magukat programozóknak.

Egy DSL nyelv készítésének főbb momentuma, ha a követelmények adottak, a nyelvben használt szintaxis létrehozása. Egy metamodellt kell létrehozni, mely megszabja, milyen modellelemeket írhatunk a DSL nyelvünkben. Mind a metamodell, mind a nyelvben elkészített program elképzelhető grafikus (ábrák, diagramok) vagy szöveges formában. Mindkét esetben meg kell határozni az elemek formátumát, szöveges esetben a használt jelöléseket, grafikus esetben az alakzatokat. A modellelemek formátumára vonatkozó megkötéseket együttesét nevezzük a nyelv nyelvtanának.

A nyelvtan elkészítésével létrejött a nyelvünk és készen áll, hogy programokat írjunk benne. Egy program megírásakor a DSL-nek először elemeznie kell a kódot (parsing), szintaktikai analízist kell végeznie. Megvizsgálja, hogy a megírt kód, pontosabban modell megfelel-e a korábban megadott szintaxisnak. Egy ilyen elemző modult (parser) körülményes megírni, de már több olyan program is létezik, melyek képes működő elemzőt készíteni adott nyelvtanhoz. Ezeket parser generator-nek vagy compiler-compiler-nek neveznek; a Java világában a legismertebb ilyen eszköz az ANTLR.

Mivel a DSL a modellezés egyik eszköze, egy adott DSL nyelv használatakor sok lehetőségük van a fejlesztőknek, hogy a lekódolt információkból valamilyen szöveges formátumot generáljanak. Ez a formátum általában futtatható kódot jelent, de akár ember számára értelmes szöveget vagy a szoftver által felhasználható paramétereket is szokás generálni. Ezt a műveletet nevezzük kódgenerálásnak.

DSL esetében a kód helyességének ellenőrzése után történhet kódgenerálás. A DSL fejlesztését támogató környezetek a DSL nyelvben megírt programot, mint modellt leképzik egy általános célú programozási nyelvre, például C#-ra vagy Java-ra. Ebben a

lépésben már tényleges kódgenerálás történik. Azonban az esetek egy részében, a generált kód nem célnak megfelelő, hiszen az csupán a DSL nyelvben készült modell reprezentációja, további átalakítás, további kódgenerálás szükséges ezt a köztes kódot felhasználva. Természetesen, a köztes kódgenerálás lépése elhagyható, ha nem használunk ilyen környezetet, de ez nem ajánlott, ugyanis a fejlesztést jelentősen megkönnyíti.

3 Felhasznált technológiák

A szakdolgozat elkészítése alatt a legtöbbet használt eszköz az Eclipse nevű platform volt. Az Eclipse egy széles körben használt integrált fejlesztőkörnyezet (integrated development environment, IDE), melyben alkalmazásokat fejleszthetünk Java vagy egyéb programozási nyelven. Az Eclipse fejlesztője, az Eclipse Foundation azonban az Eclipse IDE-n kívül, az Eclipse platformot is fejleszti.

Az Eclipse platfrom biztosítja az alapokat az Eclipse IDE működésének. Az Eclipse platform plug-inek formájába épül fel, vagyis személyre szabhatjuk, melyik plug-inokat szeretnénk használni. Létezik plug-in a különböző programozási nyelvekhez, pl. C/C++, Python, Ruby, de akár fejlesztést segítő plug-ineket is letölthetünk, pl. Git vagy LaTex. 2001 óta az Eclipse platform szabad, nyílt forrás kódú szoftver, jelenlegi licence az Eclipse Public License. Ennek következményeként az elérhető plug-inek száma növekszik. Az Eclipse Foundation 2006 óta minden év júniusában kiadják az Eclipse platform egy új verzióját, melyek mindig valamilyen tudományhoz kötődő elnevezést kaptak.

Modellezés szempontjából az Eclipse egyik fontos plug-inja az Eclipse Modeling Framework, röviden EMF. Az EMF projekt modellek készítését, leírását és kódgenerálást tesz lehetővé. Az EMF főbb eleme egy metamodell, amit Ecore modellnek is neveznek. Ezt az Ecore modellt más metamodellek létrehozására tudjuk használni, ebből kifolyólag, rengeteg más modellezési Eclipse plug-in az EMF-et használja alapjául.

Az egyik ilyen plug-innal foglalkozok szakdolgozatom keretein belül. Ennek a neve Xtext, és kifejezetten DSL-ek implementálására készítették. Az Xtext használható parser generátorként, de e funkción kívül sok egyébbel is rendelkezik. Legfontosabb közülük az EMF-fel való integrációja. Egy DSL nyelvtanának elkészítése után, ami legtöbb esetben egy szöveges objektum, az Xtext képes azt feldolgozni és működőképes EMF Ecore modellt készíteni belőle. Ez a művelet az Xtext legmeghatározóbb képessége, ezzel gyorsabbá és gyorsabban tanulhatóvá teszi a modellkészítés lépését, és a szoftver fejlesztésére szánt erőforrásokkal is gazdaságosabban bánhatunk. Ami még

lényegesebb, hogy egy Ecore modell meglétével az Xtext felhasználhatja az EMF egyéb funkcióit is, például a kódgenerálást.

Az Xtext és az EMF a fejlesztők számára értékes Java osztályokat generálnak, melyek az általunk írt DSL nyelvtan reprezentációja, de egyéb osztályok is készülnek. Azonban fontos megjegyezni, hogy az Xtext fő célja nem osztályok létrehozása, hanem egy Eclipse plug-in generálása, amelyben az általunk létrehozott DSL nyelvben vagyunk képesek programozni. Az Xtext által generált egyéb osztályok ehhez a személyre szabott plug-inhez tartoznak és a plug-inben történő programozást segítik. Például egyik osztály a content assistot hozza létre a DSL nyelvünkben, míg másik a refaktorálással foglalkozik.

A DSL, mint modell leképezésével létrejövő Java osztályok a DSL-ben megírt kód feldolgozását segítik. A fejlesztőkre van bízva, milyen műveleteket végeznek ezekkel az osztályokkal/objektumokkal, de gyakori módszer a kódgenerálás (vagy bármilyen dokumentum generálása, a továbbiakban a kódgenerálás magába foglalja ezeket is). Ehhez is nyújt segítséget az Xtext, osztályok és a Java nyelv kódgenerálás központú változatával, melynek neve Xtend.

Egy kódgeneráló programrészlet nagyon átláthatatlan tud lenni. Ugyanis rengeteg kiíratót függvényt tartalmaz, melyek argumentumaiban csakugyan programparancsok találhatóak. Ezt a problémát oldja meg egy sémanyelv használata, mely a megírt kódot valamilyen kimenetre irányítja. Egy ilyen sémanyelv az Xtextnél használatos Xtend is, melyben objektum orientált programokat lehet írni, többnyire kódgenerálás céllal. Az Xtend letisztult, „zajmentes” Javanak tekinti magát, ahol a Javaban lévő szintaxisok elhagyhatóak, például a parancs végi pontosvessző. Ezzel a kód valóban átláthatóbb lesz, de némi problémát okozhat egy adott programsor értelmezése. Érdekesség, hogy az Xtendet az Xtextben készítették el és Java nyelvet generálnak belőle.

A szakdolgozatom alatt egyéb technológiákat is felhasználtam, melyek a fejlesztéshez nem kapcsolódnak szorosan. A feladatot Windows 10 64-bites operációs rendszeren végeztem el. Verziókövetéshez Gitet és Githubot használtam. A generált C programokat Codeblocksban szerkesztettem és futtattam. Dokumentálást Microsoft Wordben készítettem.

4 Követelmények

A dolgozat keretein belül egy DSL nyelvnek kell elkészülnie és egy hozzá tartozó kódgenerátornak. A DSL-ben meg lehessen adni karakterkódolás-párokat, illetve, hogy melyik bájtsorozatot melyik másik bájtsorozattal kell helyettesíteni, hogy a két kódolás között konverziót lehessen végezni. A DSL-hez készüljön modell, és ez az Xtext Eclipse plug-inben legyen létrehozva, és egy tetszőleges kiegészítő funkciót is valósítson meg.

A kódgenerálás a DSL nyelvben megírt programokat dolgozza fel és azokból készítsen használható C forráskódot. A generált kódnak csupán a DSL-ben leírt információkat, vagyis az egyes bájtkonverziókat kell tárolnia, nem egy működő, teljes értékű alkalmazást kell generálni. Azonban készülnie kell egy C nyelvű keretprogramnak is, mely a generált kódot használja fel, és a kettő együtt az iconv POSIX szabványt valósítják meg, mely egy függvénycsoportot definiál karakterkódolások közötti konverziókhoz.

Az iconv szabvány egy struktúrát és három függvényt foglal magába. A függvények használatához egy iconv_t nevű struktúrát kell lekérnünk és felhasználnunk, mely tartalmaz minden információt, mely egy adott konvertáláshoz szükségesek. A struktúrát lekérni az iconv_open() függvénnyel lehetséges, használat után feloldani az iconv_close() függvénnyel. A konvertáláshoz az iconv() függvény használatos, amellyel egy bemeneti bufferben tárolt szöveget átkonvertálunk egy kimeneti bufferbe.

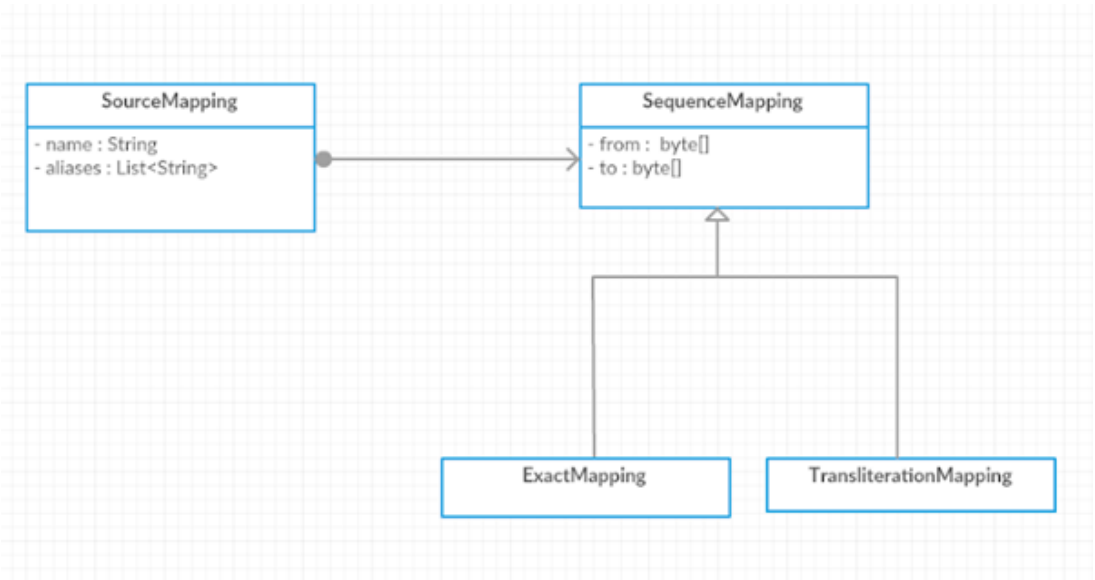
5 Tervezés

A DSL nyelvben alapvetően több karakterkódolás közötti konverziót is meg lehet adni struktúrált formában: megadjuk mi a kiinduló (source) kódolásunk, mi a cél (target) kódolásunk, majd leírjuk az egyes kódpontokat. Az egyszerűség érdekében a kódpontokat nem számmal, hanem hexadecimális kóddal írjuk le.

Itt látható egy kezdeti vázlat a DSL nyelv szintaxisához, melyben még nem szerepel a célkódolás megjelölése:

```
source someEncodingName {
  alias anAliasName

  // this is a comment

  0x0001 = 0xff01 // this will be an exact mapping
  0x0002 ~ 0xff02 // this will be a transliteration mapping
  // ...
}
```

A DSL nyelvnek, mint metamodellnek itt látható egy kezdeti diagramja:



5.1. ábra: A metamodell kezdeti diagramja

A generált C kód függvényeket tartalmazna, amellyel a konverziók adatait lehet lekérdezni. A függények nagy mértékben támaszkodnak Kövesdán Gábor hashtáblák C nyelvű implementációjára.[7] Az egyes függvények hashtáblákat adnának vissza, melyben elérhetőek az aliasok, a nyelvekben használt kódegységeket, és a mappingok.

6 Fejlesztés

6.1 A nyelvtan elkészítése:

A tervezés után a saját DSL nyelvemet kellett elkészítenem. Ennek a műveletnek a lényege, hogy Xtext-ben objektumokat/metamodell elemeket hozok létre, melyek megfeleltethetőek a tervezett nyelv egy-egy részének. Egy új Xtext projekt létrehozása után az EncodingLang.xtext fájlban írtam le a DSL-em nyelvtanát.

Mint minden Xtextes DSL-ben, így a sajátomban is először egy olyan objektumot kellett létrehoznom, amelyben a nyelv többi elemét tudjuk tárolni. Ez az objektum hagyományosan a Modell nevet kapja. Látható, hogy a Model objektum egy elements nevű listában tárolja a SourceMapping nevű objektumokat, a csillag jelzi, hogy legalább egy darabot.

```
grammar org.xtext.example.EncodingLang with
org.eclipse.xtext.common.Terminals
generate encodingLang "http://www.xtext.org/example/EncodingLang"

Model:
     (elements+=SourceMapping)*;
```

A SourceMapping objektum megadja, hogy egy karakterátkódolásnál, mely kódolásból indulunk ki, annak milyen más elnevezései vannak, és hogyan kell konvertálni egy tetszőleges célkódolásba. A DSL-ben egy SourceMapping írásakor a source kulcsszót kell először leírni, melyet az idézőjelek jeleznek. Majd egy ID-t ami, a SourceMapping name attribútumába lesz tárolva. Az ID egy ún. terminál (lásd később). Ezután felsorolhatjuk az Aliasokat, és a Conversionöket, a Modellnél látott módszerrel leírva.

```
SourceMapping:
     "source" name=ID "{"
     (aliases += Alias)*
     (conversions += Conversion)*
     "}"
;
```

Egy Conversionben megadható, hogy mi lesz a kívánt célkódolás, és adott kiinduló kódolás esetén, az egyes kódpontok, hogy változnak meg. A name feltüntetése után írhatjuk le a konkrét konverziókat, melyek objektumainak neve Mappings és a mappings nevű listában tárolódnak el.

```
Conversion:
     "target" name=ID "{"
     (mappings += Mapping)*
     "}"
;
```

Egy Mappings egy konkrét kódpontokat párosít össze, megadva, hogy egy kiinduló karakterkódolásbeli kódpont melyik kódpontnak felel meg a célkódolásban. Kétféle Mappingot különböztetek meg. Az egyik az ExactMapping, ennek írásakor = karakter, a másik a TransliterationMapping, ebben az esetben viszont ~ karakter használatos. A két Mapping közötti különbség a jelentésben van: az ExactMapping 1-1 kapcsolatban álló kódpontokat kapcsol össze, míg a TransliteraionMapping a problémásabb párosításoknál használatos. Például, ha egy karakter nincs benne a célkódolásban, így azt egy hozzá hasonló karakterhez kell párosítani. Programozási szempontból nem kezelem másként a két Mappingot, azonban a későbbi bővítés érdekében mégis két objektumot hoztam létre. A kódban lévő INPUTCHAR, az ID-hoz hasonló terminál (lásd később).

```
Mapping:
     ExactMapping | TransliterationMapping;

ExactMapping:
     from=INPUTCHAR "=" to=INPUTCHAR;

TransliterationMapping:
     from=INPUTCHAR "~" to=INPUTCHAR;
```

A SourceMapping másik attribútuma az Aliasok listája. Egy Alias objektum a nyelvben egy alias kulcsszóból és egy ID terminálból áll.

```
Alias:
     "alias" name=ID;
```

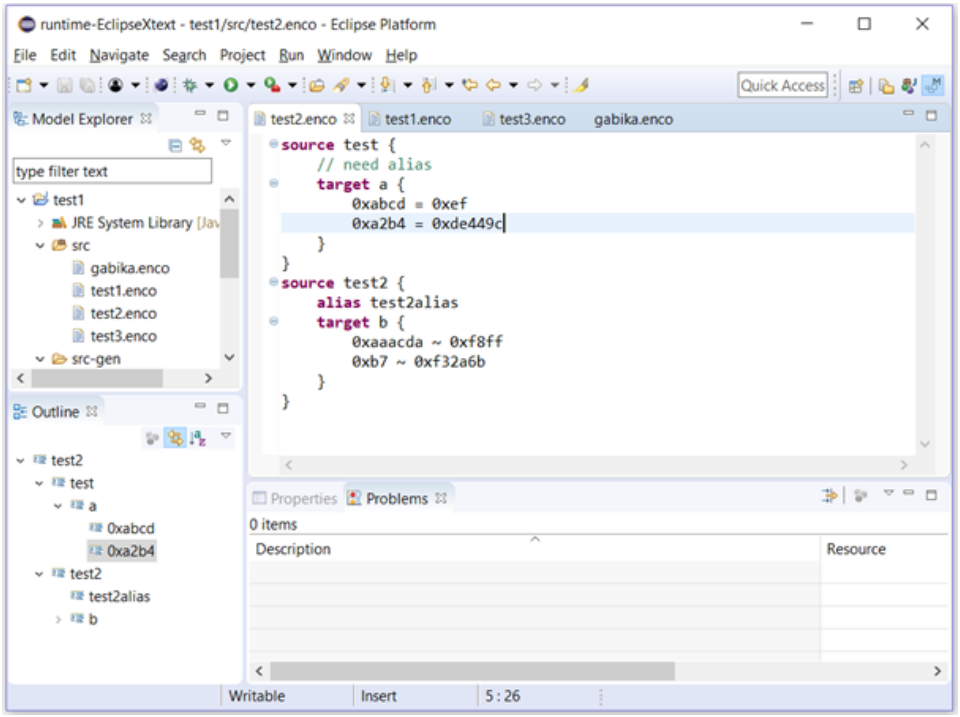
Egy nyelvtan készítésekor terminálok használata legtöbbször elengedhetetlen. A DSL nyelvünkben megírt program szöveges inputot jelent egy parser számára. A parser a kódban szereplő karakterláncoknak és sztringeknek próbál valamilyen logikai jelentést adni. A fenti objektumoknál ez a feladat nem volt nehéz a kulcsszavak miatt. De egyéb sztringeket már nem lehet objektumokhoz rendeli, mert annyira függetlenek tőlük. Ilyen például egy komment a programban. Ezeket a karakterláncokat terminálóknak nevezzük, és ezeket a terminálokat lehet hozzárendelni az objektumok attribútumaihoz. Egy terminál implementálásakor meg kell határozni, hogy mely konkrét karakterek tartoznak hozzá. Mivel a legtöbb terminál igen gyakori, az Xtextben ezek már

implementálva vannak, például az ID, az INT, a STRING, a DOUBLE, SL_COMMENT (single line comment), ML_COMMENT (multi-line comment) vagy a WS (white space). Ezen beépített Xtext terminálok automatikusan importálódnak a nyelvtanba. Kis gyakorlást igényel, hogy egy fejlesztő meg tudja határozni egy nyelv készítésekor, hogy mely nyelvi elemekhez kell a fentiekhez hasonló objektumot vagy terminált létrehoznia. Én úgy tudtam értelmezni, hogy az objektumok információkat tárolnak változókban és listákban, a terminálok pedig maguk az információk: egy integer vagy egy sztring.

Az én DSL nyelvem készítésekor egy új terminált is létre kellett hoznom, melynek neve INPUTCHAR lett. A terminál segítségével a karakterkódolásokban használt kódpontokat lehet hexadecimális formában leírni. Egy ilyen karakterlánc egy nullás karakterrel kezdődik, x-szel folytatódik, majd utána páros számú hexadecimális számrendszerbeli karakter következik, legalább két darab, kis és nagybetű között nem teszek különbséget. Például az ASCII hetvenötös kódpontú ’K’ karaktere 0x4B, vagy akár 0X4b formában is leírható.

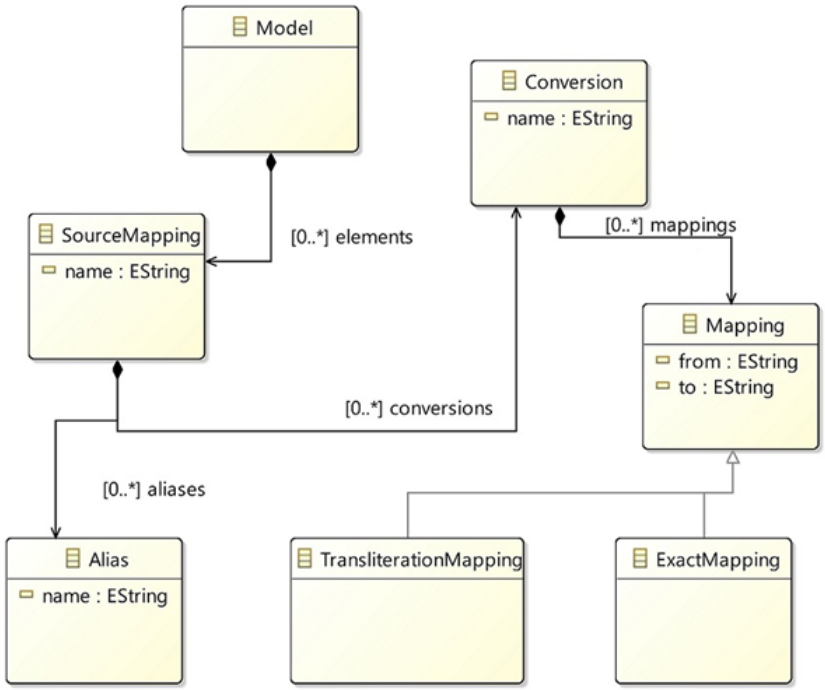
```
terminal INPUTCHAR:
'0'('x'|'X')(('0'..'9'|'a'..'f'|'A'..'F')('0'..'9'|'a'..'f'|'A'..'F'))+
;
```

Ezzel elkészült a DSL nyelv nyelvtana. Az Xtext a nyelvtanból képes egy Eclipse plug-int generláni, melyet futtatva lehetőségünk van a saját DSL nyelvünkben kódokat írni. Az Xtext egyéb funkciókat is biztosít a plug-inhoz, mely funkciók egy átlagos programozási nyelvtől vagy programozási nyelv szerkesztőtől elvárható, például a szintaxis kiemelése vagy a forráskód outlineja. A nyelv neve EncodingLang lett, a forráskódok fájlok kiterjesztése pedig .enco.



6.1. ábra: A DSL nyelv saját Eclipse plug-in-ja

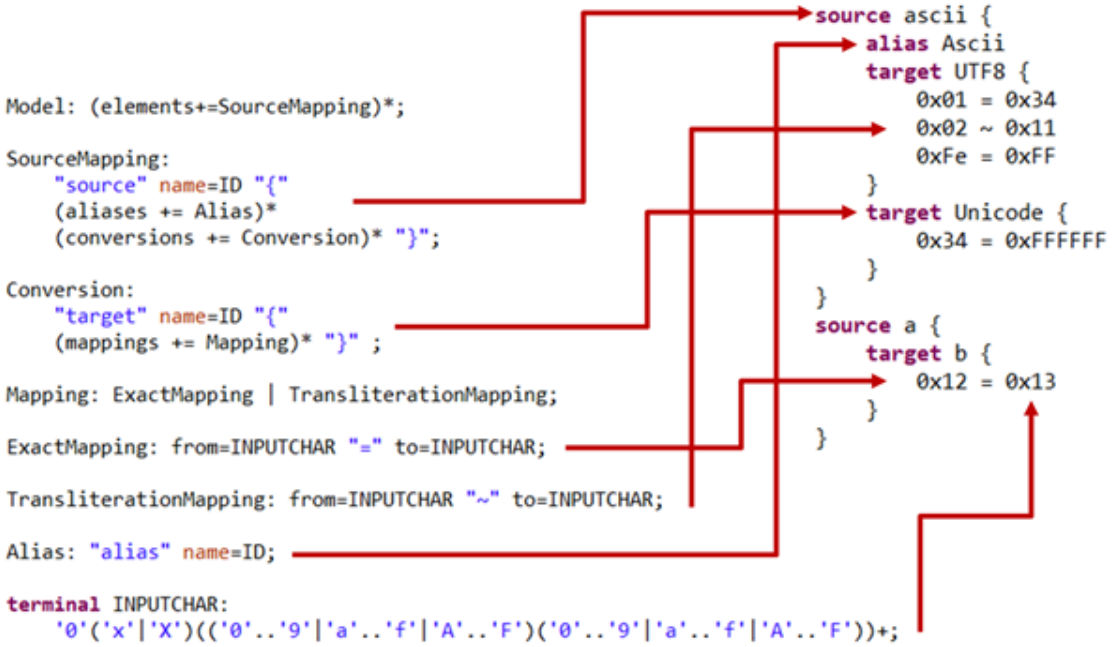
A nyelvtan elkészítéséhez használt programnyelv az Xtext saját nyelve.[8] Ebből a nyelvtani leírásból az Xtext képes EMF modellt generálni, és sok jelentős Java osztályt készíteni. Az EMF-beli Ecore modellhez volt lehetőségem egy nézetet készíteni, melyben grafikusan látható a modell diagramja.



6.2. ábra: A metamodell egy nézete

A diagram tartalmazza az összes implementált nyelvtani objektumot, és a hozzájuk tartozó attribútumokat. Ezek már ténylegesen elérhető és használható Java osztályok. Fontos megjegyezni a Mapping osztályt. Benne ugyanis nem volt deklarálva semmilyen attribútum, de mivel egy Mapping objektum vagy ExactMapping vagy TransliterationMapping lehet, és mivel mindkét leszármazott tartalmazta a from és a to attribútumokat, ezek az attribútumok ki lettek emelve az ősosztályba. Ezt, és a többi tervezői döntést az Xtext önállóan, emberi beavatkozás nélkül hozta meg.

Az Xtext objektumok és a nyelvtani elemek a következőképp állíthatóak kapcsolatba:



6.3. ábra: A nyelv és a nyelvtan kapcsolata

6.2 A nyelv tesztelése

A nyelv elkészítésével az Xtext lehetőséget biztosít a nyelv tesztelésére Java osztályokon keresztül. Kétféle szempont szerint lehet tesztelni. Az első a szintaxis ellenőrzése, vagyis, hogy a parser egy megírt programban az elvárásoknak megfelelően ismeri-e fel az egyes nyelvtani elemeket. Egy ilyen teszt teszteseteinek működése abból áll, hogy a parser bemeneteként megadunk egy helyes vagy helytelen szintaxissal rendelkező programkódot, és megvizsgáljuk, hogy a parser elfogadja vagy visszautasítja a kódot, és ebből következtetünk a parser megfelelő működésére.

A következőkben néhány tesztesetet szeretnék bemutatni példaként. Ezek a tesztek Xtend nyelven íródtak meg, ezért idegennek tűnhetnek, de értelmezésük nem

okozhat problémát. Az Xtend nyelvet a kódgenerátor implementálásánál fogom bővebben ismertetni.

Az első teszteset kezdetleges, csupán egyetlen source objektumot hozok létre benne. A parseHelper a parser egy példánya, ennek a parse() metódusa dolgozza fel a kapott forráskódot. Az Assert osztály segítségével megvizsgálhatjuk a függvény visszatérési értékét és forráskódban lévő hibákat.

```
@Test
	def void simpleSource() {
		val result = parseHelper.parse('''
			source test{}
		''')
		Assert.assertNotNull(result)
		Assert.assertTrue(result.eResource.errors.isEmpty)
	}
```

Ebben a tesztesetben a kommentezés terminálját vizsgáltam, és annak integritását a nyelvtanommal. A teszt elve ugyanolyan, mint az előző tesztesetnél.

```
@Test
	def void comment() {
		val result = parseHelper.parse('''
			source test{ //comment1
				alias testAlias1 //comment2
				/*commentline1
				  commentline2*/
				alias TestAlias2
			}
		''')
		Assert.assertNotNull(result)
		Assert.assertTrue(result.eResource.errors.isEmpty)
	}
```

A következő teszteset szándékosan egy hibás forráskóddal használja a parsert, és az elvárt eredmény is a parser hibajelzése.

```
@Test
	def void mixed_2Bytes_NumOnly() {
		val result = parseHelper.parse('''
			source test{
				alias TestAlias1
				0x1468=0x3487
				alias TestAlias2
				0x3537~0x5367
				0x8648~0x2946
			}
		''')
		Assert.assertNotNull(result)
		Assert.assertFalse(result.eResource.errors.isEmpty)
	}
```

Az utolsó teszteset, amit be szeretnék mutatni, egy olyan teszteset, melyben a INPUTCHAR terminálomat vizsgálom meg.

```
@Test
     def void multipleMapping_2Bytes() {
          val result = parseHelper.parse('''
               source test {
                    target a {
                         0xabcd=0xefef
                         0xa2b4=0xde9c
                         0xaaaa~0xffff
                         0xb78c~0xf32a
                    }
               }
          ''')
          Assert.assertNotNull(result)
          Assert.assertTrue(result.eResource.errors.isEmpty)
     }
```

A nyelvtan és a hozzá tartozó szintaxis tesztek implementálása alatt a Test Driven Development (TDD) fejlesztési módszert próbáltam ki. Ennek lényege három különböző lépés, más néven szakasz ismételt használata. Először teszteket készítek a program egy minimális funkciójáról, ez a fejlesztés piros szakasza. A következő a zöld szakasz, amikor ezt a funkciót implementálom, hogy működőképes legyen és átmenjen a teszten. Végül a kék szakasz jön, mikor is a funkciót refaktorálom, hogy a forráskód szebb és praktikusabb legyen, miközben a teszteken továbbra is átmegy. Ezt a három szakaszt az implementálás végéig folyamatosan iterálom.[9]

Számomra hasznosnak bizonyult ez a fejlesztési módszer, ugyanis adott pillanatban céltudatosan tudtam, hogy a nyelv melyik részét szeretném elkészíteni. Továbbá, a fejlesztés végeztével, a nyelvtan elkészültével jelentős mennyiségű teszteset állt rendelkezésemre. A fent bemutatott tesztesetek a TDD különböző iterációiban készültek el: például a legelső teszt a fejlesztés legelején, az utolsó pedig a legvégén.

A másik módszer, mely alapján lehetőségünk van az elkészített Xtext nyelvtant tesztelni, az Xtext által generált modellt ellenőrzi. Ha DSL nyelvtant metamodellként kezeljük, akkor a nyelvben megírt programok a metamodell példányosításainak felelnek meg. Ezek a modellek a nyelvtanban definiált objektumokból épültek fel, és az Xtext generálja őket a programkód feldolgozásával. Azonban ellenőrizni kell, hogy a modellben lévő objektumok tulajdonságai és kapcsolatai megfelelnek a programozó szándékának.

Ennek a tesztnek az alapgondolata, hogy egy használható, teljes értékű, hibamentes programkódot megírunk, átadjuk az Xtext-nek, hogy feldolgozza, majd a generált modellt több szempontból megvizsgáljuk. A vizsgálatban az objektumok attribútumait, listáit, listáinak méreteit, stb... kérjük le, és összevetjük a program írójának igényeivel, és remélhetőleg a modell megfelel az elvárásoknak, vagyis a nyelvtan helyesen működik.

Ez a teszt is Xtend nyelvben készült és a parser egy példányát használja. A programkódra meghívjuk a parse metódust, ami visszatér egy modell változóval. Majd az Assert osztály segítségével ellenőrizhetjük a generált modellt.

```
@Test
def modelTest(){
     val model = '''
     source ascii {
          alias Ascii
          target UTF8 {
               0x01 = 0x34
               0x02 ~ 0x11
               0xFe = 0xFF
          }
          target Unicode {
          }
     }
     source a {
          target b {
               0x12 = 0x13
          }
     }
     source c{}
     '''.parse

     Assert.assertNotNull(model);
     Assert.assertTrue(model.eResource.errors.isEmpty)
     Assert.assertEquals(model.elements.size, 3);

     // …

     Assert.assertEquals(model.elements.get(1).name, "a")
     Assert.assertEquals(model.elements.get(1).conversions.size, 1)
     Assert.assertEquals(model.elements.get(1).conversions.get(0).name,
"b")
Assert.assertEquals(model.elements.get(1).conversions.get(0).mappings.size
, 1)
Assert.assertEquals(model.elements.get(1).conversions.get(0).mappings.get(
0).from, "0x12")
     Assert.assertEquals(model.elements.get(1).conversions.get(0).mappings
.get(0).to, "0x13")

     Assert.assertEquals(model.elements.get(2).name, "c")
     Assert.assertEquals(model.elements.get(2).conversions.size, 0)

}
```

Ezzel a kétféle teszttel biztosra mehetünk, hogy a parser meg tudja különböztetni a szintaktikailag helyes és helytelen programkódokat, illetve, hogy a megfelelő modellt készíti el adott forráskód feldolgozásakor. Ha szükséges, további teszteket is lehet készíteni a modell egyéb kritériumaihoz, például függőségek esetén megvizsgálható, hogy nincs-e objektumok között körkörös függőség.

6.3 Egyéb programozási funkció

Szakdolgozatom részeként meg akartam ismerni azon kiegészítő funkciókat, amelyekkel bővíteni tudom a DSL nyelvemet vagy a nyelv szerkesztőjét. Ezen funkciók megkönnyítik a nyelvben való programozást, de egyéb előnyeik is lehetnek. Ilyen funkció például a content assist, scoping, sorosítás, quick fixes vagy az Xtext által már valamilyen mértékben kidolgozott szintaxiskiemelés. Ezen kiegészítések implementálásához az Xtext Java osztályokat szolgáltat és a plug-inbe való integrációját is megoldja.

Számomra a nyelv formázása tűnt a legérdekesebbnek. A formázás egységesíti a forráskódban lévő whitespaceek használatát, javítja a forráskód elrendezését és olvashatóbbá, átláthatóbbá teszi a kódot. Sőt, akár a programozási hibák számát is csökkentheti, azáltal, hogy az olvashatóbb kódban könnyebben tudjuk a hibákat felfedezni. Továbbá, az egységesített kódok miatt, különböző forráskódokban lévő eltéréseket könnyebben meg lehet találni (git diff). A szerkesztőfelületek általában segítenek, hogy automatikusan lehessen formázni a programot, például billentyűkombinációval. Azért választottam a formázás funkcióját, mert nem tudtam elképzelni, mennyire egyszerű vagy monoton feladat egy nyelv kinézetét meghatározni, de szerencsére nem volt se hosszú, se unalmas tevékenység.

Az Xtext ehhez a funkcióhoz is generált Java osztályt, melyben Xtend nyelven kellett programozni. A formázás implementálásakor a nyelvben található ún. kulcsszavakból kellett kiindulnom. Ezek a kulcsszavak a nyelvtanban található sztringek (nem a STRING terminál és nem terminálok), melyeket program írása során bele kell írni a kódba. Az én DSL nyelvemben nincs túl sok kulcsszó: source, target, alias, {, }, =, ~. Formázás során bemenetként megkapom a program modelljét, vagyis a DSL nyelvtan példányosítását, és az ebben található objektumokhoz vagy objektumok kulcsszavaihoz rendelhetek whitespaceket. Többnyire az elemek elé vagy után tuduk beszúrni whitespacet, de például beállítható, hogy egy nyitó és záró kapcsos zárójel közötti sorok

egy tabulátornyi hellyel bentebb kezdődjenek. Formázás során végig kell iterálni az összes objektumon és beállítani az egyes whitespaceket.

Az első formázó függvényt az Xtext generálta, bár a függvény üres volt. A függvény paraméterként megkapja az éppen formázni kívánt elemet, jelen esetben a Modell objektumot. Mivel a Modell csak tárolja a nyelv többi elemét, lekérjük tőle a tárolt SourceMappingokat, és mindre meghívjuk a format függvényt. Az Xtend nyelv sajátossága, hogy itt nem a sourceMapping format függvényét hívjuk meg, hanem a format függvényt, melynek egyik paramétere a sourceMapping objektum.

```
def dispatch void format(Model model, extension IFormattableDocument
document) {
	for (SourceMapping sourceMapping : model.getElements()) {
		sourceMapping.format;
	}
}
```

A következő függvény a SourceMapping formázását végzi el. A használt append függvények az elem után helyez ez whitespace-t, az első sorban például a source kulcsszó után egy szóközt helyez el. A prepend függvény az elem elé helyez whitespacet. Az interior függvény a kapcsos zárójelek között sorokat határozza meg, jelen esetben azt, hogy minden sor egy tabulátorral kezdődjön. Ezután az Aliasokra és a Conversionökre hívjuk meg a format függvényt.

```
def dispatch void format(SourceMapping s, extension IFormattableDocument
document) {
	s.regionFor.keyword("source").append[oneSpace]
	var open = s.regionFor.keyword("{")
	open.prepend[oneSpace].append[newLine]
	var close = s.regionFor.keyword("}")
	close.append[newLine]
	interior(open, close)[indent]
	for (Alias alias : s.getAliases()) {
		alias.format
	}
	for (Conversion conversion : s.getConversions()) {
		conversion.format
	}
}
```

Az Aliasok formázása nem bonyolult. Minden Alias után új sor következik, illetve az alias kulcsszó előtt egy tabulátor áll, utána pedig egy szóközt helyez el.

```
def dispatch void format(Alias a, extension IFormattableDocument document)
{
	a.append[newLine]
	a.regionFor.keyword("alias").prepend[indent].append[oneSpace]
}
```

Egy Conversion formázása megegyezik a SourceMappingéval. Mivel a Conversion tárolja a hozzá tartozó Mappingokat, ezért a Mappingok lisáját végig kell iterálni, és mindegyikre meghívni az illeszkedő format függvényt.

```
def dispatch void format(Conversion c, extension IFormattableDocument
document) {
     c.regionFor.keyword("target").prepend[newLine].append[oneSpace]
     var convopen = c.regionFor.keyword("{")
     convopen.prepend[oneSpace].append[newLine]
     var convclose = c.regionFor.keyword("}")
     convclose.append[newLine]
     interior(convopen, convclose)[indent]
     for (Mapping mapping : c.getMappings()) {
          mapping.format
     }
}
```

A Mappingok formázásánál gondom akadt, ugyanis egy adott Mapping esetén nem tudtam, hogy melyik kulcsszóhoz tudok whitespacet illeszteni. Le kellett volna valahogy kérdeznem a Mappingtól, hogy melyik fajta Mapping, hogy eldönthessem, melyik kulcsszót kell használnom (= vagy ~). Azonban tudtam, hogy ez objektum orientált programozás szempontjából kerülendő megoldás, így ezt elvetettem. Kutatás után a megoldás egyszerűbb lett, mint gondoltam. Amikor egy kulcsszót szeretnénk megtalálni az objektum programkódbeli sorai között, nem generálódik hibajelzés, ha olyan kulcsszót keresünk, amely nincs a kódban. Így meg tudtam adni mindkét féle kulcsszónak a környezetét, és amelyik szerepel a forráskódban, az fog érvényesülni.

```
def dispatch void format(Mapping m, extension IFormattableDocument
document) {
     m.append[newLine]
     m.regionFor.keyword("=").surround[oneSpace]
     m.regionFor.keyword("~").surround[oneSpace]
}
```

A formázás elkészülte és kipróbálása után egy tesztet is írtam hozzá, természetesen egy Xtext által készített osztályban. A formázás tesztelése elég egyértelmű feladat. Össze kellett hasonlítanom egy automatikusan és egy kézzel formázott forráskódot.

A teszt végrehajtásához a FormatterTestHelper osztályt biztosította az Xtext.

```
@Inject extension FormatterTestHelper formatterTester
```

Az összehasonlítást a FormatterTestHelper asserFormatter metódusa végzi, nekünk csak a bemeneti forráskódokat kell előállítanunk. Először egy formázatlan kódot adtunk meg, erre automatikusan meg lesz hívva a formázó függvény.

```
@Test
def void formatting() {
     assertFormatted[
          toBeFormatted = '''
               source test {
                                   // need alias
                    target a{
                                   0xabcd = 0xef
                    0xa2b4 = 0xde449c
                    }
               }
               source test2 {
                    alias test2alias target b {
                    0xaaacda ~ 0xf8ff
                    0xb7 ~ 0xf32a6b
                    }}
          '''
```

Másodszor, a formázás elvárt végeredményét kellett megadni.

```
          expectation = '''
               source test {
                    // need alias
                    target a {
                         0xabcd = 0xef
                         0xa2b4 = 0xde449c
                    }
               }
               source test2 {
                    alias test2alias
                    target b {
                         0xaaacda ~ 0xf8ff
                         0xb7 ~ 0xf32a6b
                    }
               }
          '''
     ]
}
```

A teszt sikeresen lefutott, ami a formázó helyes működését igazolja.

6.4 Kódgenerátor

A nyelvtan elkészülése után az Xtext generál egy Eclipse plug-int, melyben a saját nyelvünkben lehet programozni. Az Xtext továbbá egy parsert is készít, amely a DSL nyelvben megírt programokat dolgozza fel, és Java osztályokat hoz létre a DSL program modelljének megfelelően. A Java osztályokat tetszőlegesen fel lehet dolgozni, például statisztikák készítése, de a kódgenerálás is egy lehetőség.

Az Xtext által készült osztályok egyike a kódgenerátor. A generált osztályban eleinte csak minta függvényeket készít az Xtext. Megmutatja, hogyan lehet fájlokat

létrehozni, hogyan lehet Hello World programot kiírni a fájlba. Ez az osztály is Xtend nyelven íródott.

Az Xtend nyelv a kódgenerálásnál mutatja meg erősségeit. Kiemelendő a multiline sztring használata, mivel sokkal átláthatóbbá teszi a forróskódot, amelyet mi generálunk. Továbbá lehetőség van kilépni a sztringből, úgy hogy azt ne szakítsa meg. Erre a « és a » karakterek adnak lehetőséget, amelyek nincsenek benne a magyar billentyűzetekben. Két ilyen karakter között számításokat végezhetünk és függvényeket hívhatunk meg, és ezek eredményei kerülnek bele a multiline sztringbe.

Az osztály fő függvénye a doGenerate függvény, ez lesz meghívva az Xtext által. Bemeneti változói egy Resource objektum, mely a lekódolt modellt tartalmazza, egy IFileSystemAccess2, mely a fájlrendszerben képes módosításokat végezni, és egy IGeneratorContext, melyet használatára nem volt szükségem. A fsa objektum generateFile metódusával lehet a kódgenerálást elindítani. Megadjuk, hogy melyik fájlba szeretnénk kódot írni, majd azt a metódust, amely generálja a kód sztringjét. Látható, hogy két fájlt generálok, egyet a C függvényeknek, egyet a header fájlnak. A calcFileName egy saját függvény, mely a resource fájlból kiszámítja a DSL programfájl nevét, annak érdekében, hogy a generált fájlnak is ugyanaz lehessen a neve.

```
override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
     fsa.generateFile(resource.calcFileName+"_gen.txt",  resource.generate
)
     fsa.generateFile(resource.calcFileName+"_gen_header.txt",
resource.generateHeader )
}
```

Először a könnyebbik függvényt, header fájl függvényét szeretném bemutatni. Az egész függvény, a többi generáló függvényhez hasonlóan, csupán egy multiline sztring, melyet a három nyitó és három záró aposztróf jelöl. Az Xtend úgy értelmezi ezt a függvényt, hogy a sztring a visszatérési érték és az fsa.generateFile ezt a sztringet fogja fájlba írni. Így próbálja az Xtend a kódgenerálást átláthatóbbá, praktikusabbá tenni, hogy a generálandó kódot helyezi fókuszba. Továbbá megfigyelhető, a sztringből való kilépés használata is, mikor is függvényeket hívok a fájlnév kiszámítására. A generált kódban három C függvény lesz elérhető, egy az aliasoknak, egy a kódegységeknek és egy a mappingoknak.

```
def generateHeader(Resource r)'''
//Generated by HIM
#ifndef «r.calcFileName.toUpperCase»_GEN_H
```

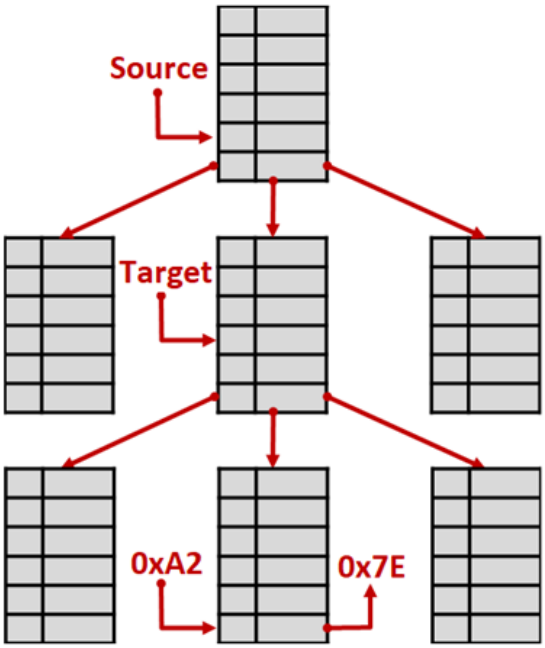
```
#define «r.calcFileName.toUpperCase»_GEN_H

#include <stdint.h>
#include <assert.h>
#include "hashtable.h"

hashtable* aliases_hashbtable();
hashtable* unit_lenghts();
hashtable* mappings();

#endif
'''
```

Ezután a C függvények implementációját generáló metódusokat mutatom be. Először egy kisebb függvény hívódik meg, mely további függvényeket hív az egyes C függvényekhez. Ezzel kicsit tagoltabb és bővíthetőbb lesz a Java kód.

```
def generate(Resource r) '''
//Generated by HIM
#include "«r.calcFileName».h"

«r.generateAliases»

«r.generateUnitLengths»

«r.generateMappings»
'''
```

A három függvény közül a mappingokhoz generáló függvényét ismertetem először. Majd a másik két függvényt, melyek komplexitása kisebb, de az elvük és a megvalósításuk hasonló.

A mappingokhoz tartozó C függvényben hashtáblák hierarchiáját kell megvalósítani. A függvény visszatérési értéke egy hashtábla lesz. Ez a hashtábla további hashtáblákat tárol, melyekben a kiinduló kódolás szerinti konverziók találhatóak. Egy kiinduló kódolás hashtáblájában ugyancsak hastblákat tárolunk a cél kódolás szerint. A cél kódolás táblájában viszont már a tényleges bájtkódok tárolódnak, melyek leírják az adott kiinduló kódolásból a célkódolásba átvivő konverziókat. Egy vázlatos ábra a hastáblák hierarchiájához:



6.4. ábra: A generált hashtáblák hierarchiája

A kódgenerálás ezen részének az elve az, hogy az Xtext által előállított Java osztályokon végigiterálunk, és az egyes objektumok attribútumait leképezzük C kódra. A függvényben először is inicializálok egy hashtáblát, mely minden egyéb hashtáblát tárolni fog. Inicializáláskor meg kell adni, hogy milyen értékeket szeretnénk tárolni benne, jelen esetben sztringekhez rendelünk hashtáblákat.

```
def generateMappings(Resource r)'''
     hashtable* mappings(){
          // hashtable for everything
          hashtable*  mappings  =  hashtable_init(128,  sizeof(char[10]),
sizeof(hashtable*));
```

Következőnek kezdődhet az elemeken való iteráció. Ehhez valamilyen ciklus lenne a legelőnyösebb, amihez az Xtend készségesen biztosít lehetőséget. Egy ciklus létrehozásához a «FOR» … «ENDFOR» művelet használható, mellyel kilépünk az multiline sztringből és például végigiterálhatunk egy listán. Ezzel rengeteg időt spórolunk meg, és mondhatni ez a «FOR» ciklus az, ami miatt a kódgenerálás folyamata előnyösebb, mint a kézzel való kódolás. Segítségével egyszerűbb és gyorsabb lesz adott kód előállítása. A resource objektum, ami a modellünket biztosítja, egyik függvényének a segítségével ki lehet szűrni adott típusú objektumokat, jelen esetben a SourceMapping elemeket a DSL program modelljéből. A hashtable_put függvény helyez el kulcs-érték párokat egy hashtáblában: ebben az esetben a fő hashtáblába helyezzük el a SourceMapping nevét, illetve a hozzá tartozó hashtáblát. A felhasznált calcString

függvény egységesíti a bemeneti sztringek hosszát, hogy könnyebb legyen a táblákban való tárolásuk.

```
«FOR s:r.allContents.toIterable.filter(SourceMapping)»
          // hashtable for «s.name»
          const char* encoding_«s.name» = "«calcString(s.name)»";
          hashtable*    mappings_from_«s.name»    =    hashtable_init(128,
sizeof(char[10]), sizeof(hashtable*));
          hashtable_put(mappings,                          encoding_«s.name»,
&mappings_from_«s.name»);
```

Értelemszerűen, ezután az adott SourceMappinghoz tartozó Conversionök következnek. Itt az Xtend egy másik nyelvi elemét, az «IF» … «ENDIF» műveletet is használom. Ennek az elágazásnak a működése abból áll, hogy, ha az IF mellé igaz értékű kifejezés kerül be, akkor a «IF» … «ENDIF» közötti részt tartalmazni fogja a sztring és így a generált kód is.

```
     «FOR c:s.conversions»
     «IF !c.mappings.isEmpty»
     // hash for «s.name»->«c.name»
     const char* encoding_«s.name»_to_«c.name» = "«calcString(c.name)»";
```

A kódnak ezen a részén, már foglalkozni kell a konverziókhoz tartozó bájt-párokkal, ugyanis ezek méretéhez kell igazítani a hashtáblák méretét. Ezek vizsgálatára szolgál a getLengthOfHash és a getIntType függvény.

```
hashtable* mappings_from_«s.name»_to_«c.name» = hashtable_init(
          «getLengthOfHash(c.mappings.get(0).from.length-2)»,
          sizeof(«getIntType(c.mappings.get(0).from.length-2)»),
          sizeof(«getIntType(c.mappings.get(0).to.length-2)»));
     hashtable_put(mappings_from_«s.name»,   encoding_«s.name»_to_«c.name»,
&mappings_from_«s.name»_to_«c.name»);
```

Itt is és a generálás többi részén is figyelni kellett arra, hogy az egyes változók nevei ne egyezhessenek meg, mert az C kód fordítása hibát eredményezne. Ennek érdekében kellett minden változóhoz belevenni a hozzá tartozó modellbeli objektum nevét.

Most már az egyes bájtsorozatokat is el tudjuk tárolni. Az egyszerűbb tárolás érdekében, nem a hexadecimális kódot tároljuk, hanem a tényleges bájtokat számok formájában. Ehhez el kellett készítenem a HexToDec osztályt Java nyelven, melynek hex2decimal függvénye ezt az átalakítást végzi el.

```
     //filling in «s.name»->«c.name»
     «getIntType(c.mappings.get(0).from.length-2)»
from_value_«s.name»_to_«c.name»;
```

```
     «getIntType(c.mappings.get(0).to.length-2)»
to_value_«s.name»_to_«c.name»;

     «FOR m:c.mappings»
     from_value_«s.name»_to_«c.name» = «HexToDec.hex2decimal(m.from)»;
     to_value_«s.name»_to_«c.name» = «HexToDec.hex2decimal(m.to)»;
     hashtable_put(
          mappings_from_«s.name»_to_«c.name»,
          &from_value_«s.name»_to_«c.name»,
          &to_value_«s.name»_to_«c.name»);

     «ENDFOR»
```

A modell bejárásával az összes hashtábla feltöltésre került. A ciklusok és elágazások lezárásával, a generált függvény visszatér a mappings nevő fő hashtáblával, a generáló függvény pedig lezárja a multiline sztringet.

```
     «ENDIF»
     «ENDFOR»

     «ENDFOR»
     return mappings;
}
'''
```

A függvények, melyek a másik két C függvényt generálják, a fentihez nagyon hasonlóan működnek. A különbség, hogy azok csak egy-egy hashtáblát készítenek és töltenek fel. Az aliasok hashtáblájához tartozó függvény a SourceMapping objektumokon iterál végig, melyet megint a resource objektumtól kér el. Az aliasok hashtáblájába a SourceMapping neve is bekerül, mely kulcshoz tartozó érték szintén a neve lesz. Így akkor is lesz eredmény, ha a SourceMapping nevét keresnénk a listába.

```
def generateAliases(Resource r)'''
     hashtable* aliases_hashbtable(){
          hashtable*   aliases   =   hashtable_init(256,   sizeof(char[10]),
sizeof(char[10]));

          «FOR s: r.allContents.toIterable.filter(SourceMapping)»
          //aliases for «s.name»
          const char* encoding_«s.name» = "«calcString(s.name)»";
          hashtable_put(aliases,encoding_«s.name»,encoding_«s.name»);
```

Ezután az egyes SourceMappingok aliases listáját bejárva elmenti az Aliasok neveit a hashtáblába. Ezzel miden lényegi adatot elmentettünk, visszatérhetünk a hashtáblával és a generáló függvényt lezárhatjuk. Itt megjegyezném, hogy az üres sorok és tabulátorok (pl a két «ENDFOR» közötti sor), illetve a generált kommentek fontos szerepet játszanak a generált kód olvashatóságában.

```
     «FOR a: s.aliases»
          //put to «s.name» alias «a.name»
          const         char*         alias_«a.name»_for_«s.name»         =
"«calcString(a.name)»";

     hashtable_put(aliases,alias_«a.name»_for_«s.name»,encoding_«s.name»);

     «ENDFOR»

     «ENDFOR»
          return aliases;
          }
'''
```

Az utolsó függvényben az egyes karakterkódolásokhoz tartozó kódhosszakat kell eltárolnunk, vagyis hány bájton tárolunk egy karaktert az adott kódolásban. Ehhez egy darab hashtáblát használok, melyben sztring kulcsokhoz rendelek egész számokat. A függvényt itt is egy hashtábla inicializálásával kezdem, majd a SourceMappingok iterálást kezdem el. Az egyes karakterkódolásokhoz tartozó kódhosszakat az első megadott karakterkonverzióból számítom ki. SourceMappingok esetén azonban egy «IF» feltétellel meg kell vizsgálni, hogy tartozik-e hozzá konverzió és abban van-e megadva leképezés. Ha ugyanis nincs, de mégis hivatkozunk az első elemekre, a generáló hibába fog ütközni.

```
def generateUnitLengths(Resource r)'''
     hashtable* unit_lenghts(){
          hashtable* unit_lengths = hashtable_init(128, sizeof(char[10]),
sizeof(uint8_t));

          «FOR s: r.allContents.toIterable.filter(SourceMapping)»
          «IF                   !s.conversions.isEmpty                    &&
!s.conversions.get(0).mappings.isEmpty»
          //unit legnths for «s.name»
          const char* encoding_«s.name» = "«calcString(s.name)»";
          uint8_t                    length_«s.name»                      =
«(s.conversions.get(0).mappings.get(0).from.length-2)/2»;
          hashtable_put(unit_lengths,                   encoding_«s.name»,
&length_«s.name»);
```

Majd a SourceMappinghoz tartozó Conversionöket nézem végig és a célkódolás kódhosszát számítom ki és mentem el. Itt is meg kell vizsgálni, hogy a Conversionhöz tartozik-e konverzió, ugyanis a nyelv definíciója szerint nem kötelező megadni hozzá.

```
     «FOR c:s.conversions»
          «IF !c.mappings.isEmpty»
          //unit length for «c.name»
          const        char*        encoding_«s.name»_to_«c.name»        =
"«calcString(c.name)»";
          uint8_t length_«s.name»_to_«c.name» =
```

```
 «(c.mappings.get(0).to.length-2)/2»;
          hashtable_put(unit_lengths,       encoding_«s.name»_to_«c.name»,
&length_«s.name»_to_«c.name»);
```

Végül lezárjuk a ciklusokat, visszatérünk a hashtáblával, a generáló függvényt pedig lezárjuk.

```
     «ENDIF»
          «ENDFOR»

          «ENDIF»
          «ENDFOR»
          return unit_lengths;
          }
'''
```

6.5 A kódgenerátor használata

A kódgenerátor használatához az Xtext által készített plug-int kell futtatnunk. A plug-inban már ismertetett módon van lehetőségünk a DSL nyelvnek megfelelő forráskódot szerkeszteni. A kódgenerálást úgy tudjuk elindítani, ha egy forráskódot elmentünk, feltételezve, hogy a program helyes. A mentés művelete tüzeli el a kódgenerálás eseményét (trigger), mikor is meghívódik a generátor doGenerate függvénye, ami két szövegfájl hoz létre melyek a generált C kódokat tartalmazzák.

Egy rövid példán keresztül szemléltetni a dolgozatom lényegi részét, a tényleges kódgenerálást. Vegyük az alábbi DSL programot:

```
source test2 {
     alias TesT
     target a {
          0xab = 0x45ef
     }
}
```

A fájl neve test2.enco. Mentés után létrejönnek a test2_gen_header.txt és a test2_gen.txt fájlok. A header fájl helyesen azt tartalmazza, amit a generáló függvényében megírtunk. Az include guardnál figyelhető meg, hogy helyesen lett kiszámítva a fájl neve.

```
//Generated by HIM
#ifndef TEST2_GEN_H
#define TEST2_GEN_H

#include <stdint.h>
#include <assert.h>
#include "hashtable.h"
```

```
hashtable* aliases_hashbtable();
hashtable* unit_lenghts();
hashtable* mappings();

#endif
```

A másik fájlban, a test2_gen.txt-ben a függvények implementációja található. Elől találjuk az aliasok függvénye. Látható, hogy lesznek egységes hosszúságúak a sztringek, mennyire átlátható a program a whitespacek miatt és hogy a változók nevei egyediek.

```
//Generated by HIM
#include "test2.h"

hashtable* aliases_hashbtable(){
     hashtable*    aliases    =    hashtable_init(256,    sizeof(char[10]),
sizeof(char[10]));

     //aliases for test2
     const char* encoding_test2 = "test2\0\0\0\0";
     hashtable_put(aliases,encoding_test2,encoding_test2);

     //put to test2 alias TesT
     const char* alias_TesT_for_test2 = "TesT\0\0\0\0\0";
     hashtable_put(aliases,alias_TesT_for_test2,encoding_test2);


     return aliases;
     }
```

A következő függvény a kódhosszak függvénye. Itt a kódhosszak értékei helyesen lette kiszámolva, ugyanis a 0xAB tárolásához egy, míg a 0x45EF tárolásához két bájtra van szükségünk.

```
hashtable* unit_lenghts(){
     hashtable*   unit_lengths   =   hashtable_init(128,   sizeof(char[10]),
sizeof(uint8_t));

     //unit legnths for test2
     const char* encoding_test2 = "test2\0\0\0\0";
     uint8_t length_test2 = 1;
     hashtable_put(unit_lengths, encoding_test2, &length_test2);

     //unit length for a
     const char* encoding_test2_to_a = "a\0\0\0\0\0\0\0\0";
     uint8_t length_test2_to_a = 2;
     hashtable_put(unit_lengths, encoding_test2_to_a, &length_test2_to_a);


     return unit_lengths;
     }
```

A karakterkonverziók függvénye a fájl legvégén található. Itt helyesen három hashtábla lett inicializálva, melyek hierarchiája is megfelelő. Az utolsó hashtáblában elhelyezett integer értékek is helyénvalóak, hiszen 0xAB = 171 és 0x45EF = 17903. Továbbá az értékeket tároló integerek mérete is helyes (uint8_t, uint16_t).

```
hashtable* mappings(){
     // hashtable for everything
     hashtable*    mappings    =    hashtable_init(128,    sizeof(char[10]),
sizeof(hashtable*));

     // hashtable for test2
     const char* encoding_test2 = "test2\0\0\0\0";
     hashtable*       mappings_from_test2       =       hashtable_init(128,
sizeof(char[10]), sizeof(hashtable*));
     hashtable_put(mappings, encoding_test2, &mappings_from_test2);

     // hash for test2->a
     const char* encoding_test2_to_a = "a\0\0\0\0\0\0\0\0";
     hashtable* mappings_from_test2_to_a = hashtable_init(
          512,
          sizeof(uint8_t),
          sizeof(uint16_t));
     hashtable_put(mappings_from_test2,                encoding_test2_to_a,
&mappings_from_test2_to_a);

     //filling in test2->a
     uint8_t from_value_test2_to_a;
     uint16_t to_value_test2_to_a;

     from_value_test2_to_a = 171;
     to_value_test2_to_a = 17903;
     hashtable_put(
          mappings_from_test2_to_a,
          &from_value_test2_to_a,
          &to_value_test2_to_a);



     return mappings;
}
```

A kódgenerátor tesztelését use casek kipróbálásával teszteltem a fenti módon. Különböző darabszámú SourceMappinget, Conversiont és Aliast tartalmazó programokból generáltam kódot és vizsgáltam meg az eredményt. Továbbá tekintettem voltam az egyes értékekre, azok határaira és a hozzájuk illő C-beli változótípusokra. A generátort ezen eredmények függvényében módosítottam, ha szükséges volt.

6.6 A keretprogram

A generált kódokhoz készítettem egy C keretprogramot, mely a felhasználja az eltárolt adatokat. A generált kód és a keretprogram együttese az iconv függvényeket

valósítják meg. Az iconv karakterkódolások közötti konverzióhoz ad lehetőséget. A iconv követelményeit és működését a keretprogramom header fájlján keresztül szeretném bemutatni.

A header fájlban először is a következő headeröket importáljuk: stdlib.h; errno.h; hashtable.h, mellyel hashtábla-műveleteket végezhetünk; és a test3.h, mely a generált kód headerje.

```
#ifndef ICONV_HIM
#define ICONV_HIM

#include<stdlib.h>
#include<errno.h>

#include "test3.h"
#include "hashtable.h"
```

A függvények használatakor egy iconv_t nevű struktúrát kell lekérnünk és használnuk, mely tartalmaz minden információt, mely egy adott konvertáláshoz szükségesek. Azonban, hiba esetén az struktúrát előállító függvénynek -1 értékkel kell visszatérnie, melyet át kell kasztolnia az iconv_t struktúrára. Ennek a problémának a megoldására létrehoztam egy másik struktúrát, mely a tényleges információkat tartalmazza és a neve iconv_t_data. Az iconv_t pedig egyetlen darab void* lesz, melyre át lehet kasztolni egy iconv_t_datat vagy egy -1-et. Ezt megoldást más iconv implementációk is használják.[10] A kívánt adatok, melyeket egy iconv_t_data struktúrában tárol, a következők: a két kódolás kódhossza, a konverziókat tartalmazó hashtábla, illetve az összes konverziót tartlamazó hashtábla, hogy használat után fel tudjuk szabadítani.

```
typedef struct {
    hashtable *allmappings;
    hashtable *mapping;
    uint8_t from_length;
    uint8_t to_length;
} iconv_t_data;

typedef void* iconv_t;
```

Ezután három függvény deklarációja jön, melyek a használatukat is sejtetik. Először az iconv_open függvénnyel lekérünk egy iconv_t-t adott kiinduló- és célkódoláshoz. Ezután az iconv függvénnyel átkonvertáljuk a kívánt buffereket. Végül a lezárjuk az iconv_t struktúrát az iconv_close segítsgégével.

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

```
int iconv_close(iconv_t cd);
size_t iconv(iconv_t cd,
    char **restrict inbuf,  size_t *restrict inbytesleft,
    char **restrict outbuf, size_t *restrict outbytesleft);
#endif // ICONV_HIM
```

A header fájl végére érve a függvények implementációját ismertetem. Értelemszerűen, az első függvény az incov_open. Ez a függvény használja fel a generált kódot. Egyesével lekéri a hashtáblákat, és a hashtáblákból eléri a kívánt adatokat. Ha hibás bemenet miatt egy hashtáblában nem találja meg a kívánt értéket, hibát jelez. Legelőször a kiinduló kódolásnak keressük meg a megfelelő elnevezését. A calcString függvény átalakítja a sztringeket, hogy egységnyi legyen a hosszuságuk.

```
iconv_t iconv_open(const char *tocode2, const char *fromcode2)
{
    char* tocode=calcString(tocode2);
    char* fromcode=calcString(fromcode2);
    iconv_t_data *returnval=(iconv_t_data*) malloc(sizeof(iconv_t_data));
    char *fromcode_realname= (char*) malloc(sizeof(char[10]));
    hashtable *aliases = aliases_hashbtable();
    if( hashtable_get(aliases,fromcode,fromcode_realname)
            == HASH_NOTFOUND )
        {errno=EINVAL;return ((iconv_t)-1);}
    hashtable_free(aliases);
```

Ezután az egyes kódhosszakat állítjuk be.

```
hashtable* lengths = unit_lenghts();
    if(     hashtable_get(lengths,     fromcode_realname,     &(returnval-
>from_length))
            == HASH_NOTFOUND ||
            hashtable_get(lengths, tocode, &(returnval->to_length))
            == HASH_NOTFOUND)
        {errno= EINVAL;return (iconv_t) -1;}
    hashtable_free(lengths);
```

Legvégül a konverziónak a hashtábláját keressük ki a rekordok közül.

```
returnval->allmappings = mappings();
    hashtable *frommapping;
    if(                                          hashtable_get(returnval-
>allmappings,fromcode_realname,&frommapping)
            == HASH_NOTFOUND)
        {errno= EINVAL;return (iconv_t) -1;}
    if( hashtable_get(frommapping, tocode, &(returnval->mapping))
            == HASH_NOTFOUND)
        {errno= EINVAL;return (iconv_t) -1;}

    return (iconv_t)(void*)returnval;
}
```

Következő függvény az iconv. Ez a függvény egy bemeneti buffert konvertál át az adott iconv_t szerint, és az átalakított bájtokat egy kimeneti bufferbe menti el (a

kimeneti buffer is egy paramétere a függvénynek, csupán a neve az, hogy kimeneti). Először is megvizsgálja, hogy használhatóak-e a bemeneti paraméterek.

```
size_t iconv(iconv_t cd,
             char **restrict inbuf,  size_t *restrict inbytesleft,
             char **restrict outbuf, size_t *restrict outbytesleft)
{
    if(cd==(iconv_t)-1)
        {errno= EBADF;return (size_t)-1;}
    iconv_t_data *con=(iconv_t_data*) cd;
    if(*outbytesleft<con->to_length)
        {errno = E2BIG;return (size_t)-1;}
```

Ezután egy ciklus következik, mely az átalakítást végzi. Az átalakítás első lépése, hogy kiveszünk annyi bájtot a bemeneti bufferből, amennyi a kiinduló kódolás kódhossza. Ehhez bitműveleteket használok.

```
    do
    {
        fromvalue=(*inbuf)[0];
        for(int i=2; i<=con->from_length; i++)
            fromvalue=(fromvalue<<8)+(*inbuf)[0+i-1];
```

Második lépésként a hashtáblából lekérdezem a bemeneti értékhez tartozó kimeneti értéket. Ha ilyen kulcshoz nincs érték a táblában, akkor a kimeneti érték ugyanaz lesz, mint a bemeneti.

```
        if(hashtable_get(con->mapping,&fromvalue,&tovalue)==HASH_NOTFOUND)
            tovalue=fromvalue;
```

Majd a kimeneti értékeket bájtonként bemásolom a kimeneti bufferbe, szintén bitműveletek segítségével.

```
        char b;
        for(int i=1; i<=con->to_length; i++)
        {
            b=(tovalue&(0xFF<<((con->to_length-i)*8)))>>((con->to_length-
i)*8);
            **outbuf=b;
            *outbuf=*outbuf+sizeof(char);
        }
```

Az iconv definíciója szerint, a bufferek pointereinek a függvény végén az utolsó sikeres konverzióra kell mutatniuk. Vagyis megvizsgálom, hogy lesz-e még sikeres konverzió, pontosabban lesz-e elég hely a bufferekbe, és ha igen, módosítom a pointerket.

```
    if(*inbytesleft>=con->from_length && *outbytesleft>=con->to_length)
        {
            *inbuf=*inbuf+con->from_length;
```

```
            *inbytesleft=*inbytesleft-con->from_length;
            *outbytesleft=*outbytesleft-con->to_length;
        }
    }
Végül lezárom a ciklust és a függvényt.
    while(*inbytesleft>=con->from_length       &&       *outbytesleft>=con-
>to_length);
    return 0;
}
```

Az utolsó függvény az iconv_close, mely az icon_t által foglalt memóriát szabadítja fel. A függvény ellenőrzi a bemeneti iconv_t változó helyességét, majd átkasztolja iconv_t_data típusúvá, és a konverziók fő hashtábláját felszabadítja a hashtable_free függvény segítségével.

```
int iconv_close(iconv_t cd)
{
    if(cd!=(iconv_t)-1)
    {
        iconv_t_data *x = (iconv_t_data*) cd;
        hashtable_free(x->allmappings);
    }
    return 0;
}
```

Az iconv függvényeim ismertetése után egy rövid példán keresztül szeretném bemutatni a használatukat. Készítsünk a DSL nyelvben egy programot, mely olyan konverziót ír le, ahol a kis e betűt nagy E betűre, az x betűt pedig 0 karakterré kell konvertálni. Egy ilyen program például a következő: (e=0x65, E=0x45, x=0x78, 0=0x30)

```
source ascii{
     alias Ascii
     target UTF8{
          0x65=0x45
          0x78~0x30
     }
}
source a{
     target b{
          0x45=0x45BA20
     }
}
```

A DSL programból generált kódot includeoljuk az iconv függvényekhez. Az iconv függvényeket a fent ismertetett módon és sorrendben meghívjuk. A konvertálandó sztring a híres „the quick brown fox jumps over the lazy dog”.

```
    const char *kiindulo = "Ascii";
    const char *cel = "UTF8";
    iconv_t cd = iconv_open(cel,kiindulo);
```

```
char *inbuf="the quick brown fox jumps over the lazy dog";
size_t inbytesleft=strlen(inbuf)+1;
char *outbuf=(char*) malloc(sizeof(char[44]));
char *outbuf2 = outbuf;
size_t outbytesleft=sizeof(char[44]);
iconv(cd,&inbuf,&inbytesleft,&outbuf,&outbytesleft);

printf("%s\n",outbuf2);
iconv_close(cd);
```

A program lefutása után a kimeneten a „thE quick brown fo0 jumps ovEr thE lazy dog” sztring jelenik meg, vagyis a konvertálás sikeres volt.

6.7 Az iconv értékelése

Dolgozatom zárásaként meg szerettem volna vizsgálni, hogy az általam készített iconv milyen mértékben hasonlít egy igazi iconv programhoz. Ennek érdekében több szempontból összehasonlítottam a GNU iconv implementációjával.[11] A két programot helyesség és teljesítmény szerint külön-külön megvizsgáltam, és a kapott eredményeket összevetettem, hogy megtudhassam, mennyire működőképes és használható az én implementációm.

Először is a helyes működést akartam megvizsgálni. Hogy ne legyen túl bonyolult a mérés, csak magyar nyelvű szövegek kódolását vizsgáltam. Kiinduló kódolásnak az ISO-8859-2 karakterkódolási szabványt választottam. Ez a kódolás 8 biten tárol kódpontokat és ASCII alapú, vagyis a kódolás a 0x7F karakterig megegyezik az ASCII kódtáblával és csak az azutáni értékeknek rendel új karaktereket. Gyakorlatban ez azt jelenti, hogy ha a bájt első bitje nulla, akkor a karakter ASCII karakter, ha pedig a bit egyes, akkor valamilyen különlegesebb karakterről van szó. Az ISO-8859 szabványok, mind ilyen elven működnek, a különbség közöttük a különleges karakterekben vannak, ugyanis e karaktereket földrajzi területek szerint válogatták csoportokba. Az általam használt ISO-8859-2 a kelet európai országok karaktereit tartalmazza (például a magyar ű betű = 0xFB), de pl. az 5-ös sorszámú a görög (ћ = 0xFB), míg a 11-es a thai karaktereket (๛ =0xFB) foglalja magába. Összesen 16 különböző ISO-8859 szabvány van. Az ISO-8859-2 másik használatos elnevezése a Latin-2.

A konverzió cél kódolásának az ASCII-t választottam. Ahhoz, hogy egy tipikus magyar szöveget ISO-8859-2 kódolásból ASCII-vé konvertáljunk, csupán az ékezetes magyar betűket kell leképezni valamelyik ASCII karakterré, logikus megoldás a nem

ékezetes karakterré leképezni. A konverzióhoz a következő programot készítettem el a DSL nyelvemben:

```
source iso88592 {
     alias latin2
     target ascii {
          // kisbetűk
          0xe1 ~ 0x61 // á
          0xe9 ~ 0x65 // é
          0xed ~ 0x69 // í
          0xF3 ~ 0x6f // ó
          0xF6 ~ 0x6f // ö
          0xF5 ~ 0x6f // ő
          0xFA ~ 0x75 // ú
          0xFC ~ 0x75 // ü
          0xFB ~ 0x75 // ű
          // nagybetűk
          0xC1 ~ 0x41 // Á
          0xC9 ~ 0x45 // É
          0xCD ~ 0x49 // Í
          0xD3 ~ 0x4f // Ó
          0xD6 ~ 0x4f // Ö
          0xD5 ~ 0x4f // Ő
          0xDA ~ 0x55 // Ú
          0xDC ~ 0x55 // Ü
          0xDB ~ 0x55 // Ű
     }
}
```

A generált forráskódot hozzáadtam az iconv függvényeimhez és már készen is állt a használatra. Egy gyors teszttel meggyőződtem a helyességéről.

Konvertálás előtt	Konvertálás után
abcdefxyz	abcdefxyz
ABCDEFXYZ	ABCDEFXYZ
áéíóöőúüű	aeioooouuu
ÁÉÍÓÖŐÚÜŰ	AEIOOOUUU

Ahhoz, hogy össze tudjam hasonlítani az én iconv-m és a GNU-féle iconv programok végeredményeit, ki kellett egészítenem a programomat, hogy az fájlból olvasson be sztringet, azt konvertálja le az iconv segítségével, majd fájlba írja ki a konvertált buffert. A megírt C program forráskódja:

```
const char *source = "iso88592";
    const char *target = "ascii";
    iconv_t cd = iconv_open(target,source);

    if(cd!=-1){
        FILE *file_in=fopen("from.txt","r");
        FILE *file_ou=fopen("to.txt","w");
```

Ennél a résznél arra kellett figyelmet fordítanom, hogy az iconv megváltoztatja a bufferek pointereinek helyzetét. Hogy ne kelljen folyton új buffereket foglalnom a memóriában, elmentem a bufferek pointereit, hogy később visszaállíthassam a bufferek elejére őket. Látható, hogy a fájlból százasával konvertálom át a karaktereket.

```
        char *inbuf=(char*) malloc(sizeof(char[100]));
        char *inbuf2=inbuf;
        char *outbuf=(char*) malloc(sizeof(char[100]));
        char *outbuf2=outbuf;
        if(file_ou==NULL) printf("file error 2");
        else if(file_in==NULL)printf("file error");
        else while(fgets(inbuf,100,file_in)!=NULL){
            size_t inbytesleft=strlen(inbuf)+1;
            size_t outbytesleft=sizeof(char[100]);
            outbuf=outbuf2;
            iconv(cd,&inbuf,&inbytesleft,&outbuf,&outbytesleft);
            fprintf(file_ou,"%s",outbuf2);
            inbuf=inbuf2;
        }
        fclose(file_in);
        fclose(file_ou);
    }

    iconv_close(cd);
```

A fenti program segítségével, már megtörténhetett a tényleges összehasonlítás. Az eredmények összehasonlításhoz egy tipikus, átlagos magyar szöveget kellett használom és ez a szöveg Jókai Mór regényének, A kőszívű ember fiainak az első pár fejezete lett.[12] A szövegre lefuttattam az én iconv függvényeimet, majd a GNU iconv függvényét is, az utóbbit parancssorból:

```
iconv -f ISO-8859-2 -t ASCII//TRANSLIT from.txt -o to_iconv.txt
```

Ezután a diff program segítségével összehasonlítottam a két fájlt.

```
diff -s -q to.txt to_iconv.txt
```

Sajnálatosan az két fájl nem egyezett meg. A különbségek megvizsgálásával észrevettem, hogy A kőszívű ember fiai mégsem tipikus magyar szöveg. Ugyanis néha német szavak fordulnak elő benne, melyekben tartalmazzák az ä betűt. Ennek a kijavítására a DSL programba beillesztettem pár bájtkonverziót. Feltűnt, hogy egy ilyen hibának a kijavítása, mennyire egyszerű modellezés és kódgenerálás használatával.

```
0xE4 ~ 0x61 // a betű két ponttal
0xC4 ~ 0x41 // A betű két ponttal
```

Ezután újra lefuttattam a fájlok konvertálását és azok összehasonlítását.

```
diff -s -q to.txt to_iconv.txt
Files to.txt and to_iconv.txt are identical
```

A két fájl megegyezett, vagyis az én iconv-m képes olyan eredményt adni, mint egy elterjedt iconv program.

Az iconv helyességének igazolása után a teljesítményét akartam megmérni. Ehhez a bemeneti fájl méretét kellett megnövelnem, majd megmérnem mindkét program futásidejét. Remélhetőleg nem lesz jelentősen lassabb az én implementációm.

A fájl növeléséhez egyszerűen többször újra bemásoltam a fejezeteket A kőszívű ember fiaiból, ezzel megsokszorozva a méretét. A bemeneti fájl kicsit több mint 3MB volt. Mindkét program pillanatok alatt átkonvertálta a fájlt, természetesen helyes eredménnyel. A megfelelő méréshez tovább kellett sokszorosítanom a fájl méretét. Ezúttal 21MB-nyi szöveget kellett átkonvertálni. A programoknak észrevehetően több idő kellett, nagyjából egy másodperc, de így is ugyanakkor végeztek a konvertálással. Az én iconv programom úgy tűnik fel tudja venni a versenyt más programmal.

Az utolsó méréshez jelentős méretre, 105MB-re növeltem a bemeneti fájlokat. Itt már tetten érhető volt a programok gyorsasága. A GNU iconv 1.073s alatt konvertálta át a fájlt, míg az én iconv-m 4.125s alatt.

GNU iconv		iconv_him	
real	0m1.073s	real	0m4.125s
user	0m0.953s	user	0m3.313s
sys	0m0.109s	sys	0m0.281s

Az eltérés jelentős, úgy sejtem a GNU iconv gazdaságosabban használja a memóriát, jobban rendszerezi a bájtkonverziókat. Ennek ellenére, meg vagyok elégedve a programom teljesítményével, ugyanis jelentős fájlméret esetén sincs nagyságrendekkel lemaradva.

Legvégül a programban lévő kódsorok száma szerint hasonlítottam össze a két programot. Ehhez a cloc programot használtam, mely megszámolja a fájlban/mappában lévő kódok sorait.[13] Ahogy sejteni lehetett, az én iconv-m mérete csupán töredéke a GNU iconv programnak. Az én iconv-m mérete:

```
Language                files          blank        comment           code
-------------------------------------------------------------------------
C                           4            120            198            466
C/C++ Header                3             19              3             54
-------------------------------------------------------------------------
```

```
SUM:                          7            139            201            520
-------------------------------------------------------------------------------
```

A GNU iconv mérete:

```
Language                files          blank        comment           code
-------------------------------------------------------------------------------
C/C++ Header              272           2874          13730          98188
Bourne Shell               27          10307           8935          54501
m4                        109           1670           1061          18727
C                          61           1456           2691           9087
PO File                    36           1710           4507           6696
       ...
-------------------------------------------------------------------------------
SUM:                      540          18773          31184         194626
-------------------------------------------------------------------------------
```

A kódsorok számánál nyilván az is közre játszik, hogy jelenleg az én iconv-m csupán egyféle konverzióra képes (ISO-8859-2-ből ASCII-t készít), míg a GNU iconv több kódolosából tudunk választani és többségében mindkét irányba működik a konverzió.

Összességében, az kódgenerátorom és a konvertálóm együttese nem a legjobb iconv implementációt nyújtják. Azonban a DSL nyelv könnyű használata, a generált kód egyszerű üzembe helyezése, a helyes végeredmények elkészítése és a relatíve alacsony futásidő mind hozzájárulnak ahhoz, hogy az elkészült alkalmazás rugalmas, egyszerű, jól működő és a gyakorlatban felhasználható legyen.

7 Összegzés

Szakdolgozatom keretein belül az informatika több területét meg kellett ismernem és egy olyan alkalmazást készítettem, amelyben régi és új ismeretek is felhasználásra kerültek. Szakdolgozatomban kódgenerálással, karakterkódolásokkal és személyes kedvencemmel, modellezéssel is foglalkoztam. Az elkészült alkalmazás magába foglal egy DSL nyelvet, egy generátort és az iconv függvényeket. A nyelvben karakterkódolások közötti konverziókat írhatunk le. A kódgenerátorral képesek vagyunk feldolgozni és működőképes C kódot generálni egy DSL-beli programból. Az implementált iconv függvények pedig a generált C kódot használják fel.

A dolgozat elkészítésének a motivációja a modell alapú fejlesztés kipróbálása volt. Modellek segítségével gyorsabb és egyszerűbb volt a fejlesztés, ezek az előnyök a kódgenerálásnál jöttek elő leginkább. Dolgozatomban azt is vizsgáltam, hogy érdemes-e régebbi problémákat újra elővenni, és újabb, modernebb eszközökkel megoldani őket. Véleményem szerint van létjogosultsága mind a modell alapokon történő fejlesztésnek, mind a régi problémák ismételt megoldásának. Bár szerintem nem lenne túl jövedelmező vállalkozás.

Az elkészült alkalmazásban természetesen vannak továbbfejlesztési lehetőségek. Az iconv függvényeket és a generált kódot lehetne optimalizálni, hogy jobban használják az erőforrásokat. Azonban a DSL nyelvben nagyobb hiányosságok vannak. Egyes karakterkódolások állapotfüggőek, vagyis a karaktereket az is befolyásolja, hogy őket milyen egyéb karakterek előzték meg. Olyan kódolások is léteznek, amelyekben egy darab karakter által elfoglalt bájtok száma változik, például az UTF8-ban, 1-4 bájtot foglal el egy karakter. Jelenleg ilyen struktúrát nem lehet leírni a DSL nyelvben, és emiatt korlátozott a használata. Ezeket a problémákat lehetne orvosolni.

Személy szerint élveztem a dolgozat és az alkalmazás elkészítését. Sok hasznos tudást szereztem a modellezésről és e tudást későbbi tanulmányokba is fel tudom használni. A DSL nyelvek és a kódgenerálás nekem mindig is egy egzotikusabb területe volt az informatikának, melyre mindig is kíváncsi voltam. Örülök, hogy a karakterkódolás elméletét is megismertem végre, mivel ez a téma túl ismeretlen volt számomra és mindig is ódzkodtam tőle. Összefoglalva, nem bántam meg, hogy ezt a

témát választottam dolgozatom témájának és büszke vagyok mindarra, amit dolgozatom keretein belül megismertem vagy elkészítettem.

Irodalomjegyzék

[1] Unicode.org: *Unicode Character Encoding Model*, http://www.unicode.org/reports/tr17/ (revision: 2008-11-11)

[2] IETF: *The Report of the IAB Character Set Workshop* https://www.ietf.org/rfc/rfc2130.txt, (revision: April 1997)

[3] Prototpyr.io: *Mojibake*, https://blog.prototypr.io/mojibake-the-unknown-very-common-problem-with-east-asian-language-input-804191067c18, (revision: 14 January 2018)

[4] Kunststube: *What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text*, http://kunststube.net/encoding/, (revision: 27 April 2015)

[5] World Wide Web Consortium: *Who uses Unicode?*, https://www.w3.org/International/questions/qa-who-uses-unicode, (revision: 6 February 2016)

[6] The Open Group: *iconv.h*, http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/iconv.h.html, (revision: 2018)

[7] Github: *hashtable.c*, https://github.com/gaborbsd/FREC/blob/master/lib/libfrec/hashtable.c, (revision: 13 February 2017)

[8] Bettini, L.: *Implementing Domain-Specific Languages with Xtext and Xtend*, 2nd edition, ISBN 978-1-78646-496-5, Packt Publishing, 2016

[9] Beck, K.: *Test Driven Development: By Example*, ISBN 978-0321146533, Addison-Wesley Professional, 2002

[10] Github: *iconv.h*, https://github.com/git-for-windows/git-sdk-64/blob/master/mingw64/include/iconv.h, (revision: 7 March 2017)

[11] GNU: *libiconv*, https://www.gnu.org/software/libiconv/, (revision: 11 February 2017)

[12] Magyar Elektronikus Könyvtár: *Jókai Mór – A kőszívű ember fiai*, http://mek.oszk.hu/00600/00695/html/

[13] Sourceforge: *CLOC*, http://cloc.sourceforge.net/, (revision: 14 January 2017)