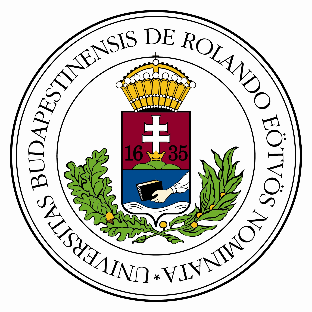
**Eötvös Loránd Tudományegyetem**

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

**P4 programok helyességének ellenőrzése bővíthető szabályrendszer alapján**

Szerző:

Nagy Rebeka

Programtervező Informatikus BSc

Témavezető:

Tóth Gabriella

Doktorandusz

Budapest, 2020

Tartalomjegyzék

[1. Bevezetés 4](#_Toc41330392)

[2. Felhasználói dokumentáció 5](#_Toc41330393)

[3. Fejlesztői dokumentáció 5](#_Toc41330394)

[3.1. Szabályrendszer 5](#_Toc41330395)

[3.2. Tervezés 5](#_Toc41330396)

[3.3. Hiba észlelés 6](#_Toc41330397)

[Parser.hs 6](#_Toc41330398)

[Preparation.hs 9](#_Toc41330399)

[Verification.hs 14](#_Toc41330400)

[Calculation.hs 21](#_Toc41330401)

[3.4. Felhasználói felület 23](#_Toc41330402)

[Model 23](#_Toc41330403)

[ViewModel 26](#_Toc41330404)

[View 27](#_Toc41330405)

# 1. Bevezetés

Egy szakdolgozat elkészítése remek lehetőség arra, hogy az egyetemi évek alatt megszerzett tudást kamatoztassuk, valamint, hogy új eszközöket és módszereket sajátítsunk el.

Mindig is közel éreztem magamhoz a funkcionális nyelveket, így a projektem modelljét is abban szerettem volna implementálni. A felhasználó felületek elkészítésére sok módszert tanultam különböző kurzusok alkalmával, ezek közül terveztem választani a szakdolgozatomhoz is egy architektúrát.

A P4 programozázi nyelvvel néhány projektmunka alkalmával ismerkedtem meg.

Ez egy napjainkban fontos, hálózati csomagok feldolgozására szolgáló eszközök programozására készített programozási nyelv. Az ellenőrzése és a tesztelése ebből kifolyólag nehéz, a fejlesztők könnyen hibát tudnak véteni a fejlesztés során.

A témám tehát egy olyan komplex szoftver elkészítése, amely képes ellenőrizni ezeket a programokat, és valamilyen visszajelzést adni arról, hogy mely részek okozhatnak nem elvárt viselkedést.

A szakdolgozatom ez alapján két nagyobb részre osztható: az elemző részre, és megjelenítésre.

Előbbihez tartozik a P4 szoftver szintaktikus elemzése, az előkészítése, végül az ellenőrzése adott szabályrendszer alapján. Ez a rész Haskell funkcionális nyelven írodott, így lehetőségem volt elmélyíteni a nyelvről megszerzett tudásomat, új könyvtárakat, implementálási módszereket és nyelvi szerkezeteket ismerhettem meg.

Utóbbit az egyetemen elsajátított MVVM architektúrában építettem fel, ahol az üzleti logika az ellenőrző résztől kapott infomációt dolgozza fel.

# 2. Felhasználói dokumentáció

....

# 3. Fejlesztői dokumentáció

## 3.1. Szabályrendszer

A verifikáció működésének a Tóth Gabriella és Tejfel Máté által szerzett cikk[[1]](#endnote-1) az alapja. Ez egy axiomatikus szemantikához hasonló szabályrendszer, amelynek összetettebb a környezeti struktúrája, és mellékfeltételekkel van kiegészítve.

Ezeket a feltételeket tudja a felhasználó módosítani, ezzel bővítve a programon végzett ellenőrzéseket.

....

## 3.2. Tervezés

A dolgozat célja egy összetett verifikációs szoftver elkészítése, amely képes egy szintaktikailag helyes, leforduló P4 résznyelvbeli programot ellenőrizni, egy adott szabályrendszer alapján.

....

A programnak könnyen használhatónak, és átláthatónak kell lennie. Fontos a helyes és gyors működés is.

Mivel az ellenőrzés és a felhasználói felület két különböző nyelven és keretrendszerben van implementálva, így a köztük lévő gördülékeny kommunikációt is meg kell valósítani. Ez úgy valósult meg, hogy a Haskellben megírt ellenőrző rész egy .dll kiterjesztésű állománnyá van alakítva. Így a C# nyelven megírt felhasználói felület modell része könnyen eléri a számításokat végző függvényt.

## 3.3. Hiba észlelés

A hiba észlelés rész Haskell nyelven lett implementálva Visual Studio Code fejlesztőkörnyezetben.

Az ellenőrzés két szöveget kap a felhasználói felülettől, ezeket dolgozza fel, majd a belőlük kiszámított fontos információkat küldi vissza, szintén szövegként.

A megkapott stringek közül az egyik maga a P4 program, a másik a felhasználó által megadott mellékfeltételek.

Négy állományból áll, melyek az elemző folyamatokat végző *Parser.hs*, az átalakító függvényeket tartalmazó *Preparation.hs*, az ellenőrzést végző *Verification.hs* és az ezeket összefogó és a felhasználói felülettel kommunikáló *Calculation.hs*.

### Parser.hs

Az első lépésben a program feldolgozása történik, amelyet a Text.ParserCombinators.Parsec[[2]](#endnote-2) könyvtár, valamint egy egyszerűbb példa[[3]](#endnote-3) segítségével oldottam meg.

A nyelvtani elemzésehez szükség van a nyelv definíciójára, vagyis a kommentek jelölésének, a változónevek leírásának, az operátorok és a kulcsszavak meghatározására. Ezzel az is definiálásra kerül, hogy a P4 milyen résznyelvével foglalkozik az ellenőrzés.

Implementálásra kerültek még bizonyos tokenek, amelyek magát az elemzést és annak olvashatóságát segítik.

#### Bevezetett adattípusok

##### Statement

A szintaktikus elemző célja, hogy a megkapott P4 programot megfelelő részekre bontsa, amelyet az előkészítő folyamat már könnyen át tud alakítani a hibák észleléséhez szükséges típusokra.

Ezeket a részeket a *Statement* típus foglalja magába. Az elemzés célja egy *Statement* listává alakítani a megkapott programot.

##### Variable

A *Variable* típus segítségével írtam le mind a mezőket és fejléceket, mind pedig a táblák kulcsait és a kontrollfüggvények paramétereiben szereplő változókat.

##### FunctionExpression

A programban történő függvényhívásokat *FunctionExpression*-ként írtam le, melyek a *FuncExpr* konstruktorral alakíthatóak át *Statement*-re.

Ezek a függvényhívásokhoz tartozó változókból, valamint a függvény fajtájából tevődnek össze.

##### ArithmeticExpression

Az aritmetikai kifejezések leírására szolgáló típus. Ezt a *ParserAssignment* konstruktorral használjuk, amely a programban lévő értékadásokat írja le. Ehhez tartozik egy *string*, amely az egyenlőség bal oldalán szereplő változó nevét takarja, valamint egy *ArithmeticExpression*, amelyben az értékadás bal oldalán szereplő kifejezést tároljuk. Ez állhat egyetlen változóból, vagy bármilyen aritmetikai kifejezésből (negálás, összeadás, kivonás, szorzás, osztás).

##### BoolExpression

A logikai kifejezéseket írhatjuk le vele. Ezek a résznyelvben az elágazások feltételében szerepelnek. *Statement*-re a *BoolExpr* konstruktorral tudjuk hozni.

Állhat logikai konstansokból (igaz, hamis), valamint ezek és változók valamilyen logikai kifejezéséből (negálás, konjunkció, diszjunkció, egyenlő, nem egyenlő, kisebb, nagyobb).

Egy P4 specifikus függvény, a *.isValid()* is ide tartozik.

#### Függvények

Az szintaktikus elemzőt képező függvények három nagyobb részre oszthatóak, amelyek lejjebb kerülnek kifejtésre. A *Parsec* könyvtárból származó *Parser* függvényparaméter segítségével tudjuk a függvényeket olyan alakra hozni, amellyel szöveget tudunk feldarabolni, mintát illeszteni rá és meghatározni, hogy a programnak pontosan milyen alkotóeleme. Ezek a szeletelések mindig a legnagyobb illesztést hajtják végre, így cél volt az, hogy az elemzés során mindig egyértelmű legyen, hogy mi a következő lépés.

Egy adott szöveg esetében a *parseString*, míg fájlból olvasás esetén a *parseFile* függvénnyel lehet az elemzést végrehajtani.

##### Fejléc-Struct elemző függvények

Ezek a függvények a program elején definiált fejlécekből és struktúrákból szedik ki a számunkra hasznos információkat, vagyis, maga az elnevezés mellett, a mezők neveit. A *header* és a *struct* kulcsszavak megtalálásakor indul el az ehhez tartozó feldolgozás. *ParserHeader* és *ParserStruct* konstruktorral kerülnek tárolásra.

##### Parser elemző függvények

Ez egy P4 specifikus része a programnak. Az elemzés során ebből a részből fogjuk kinyerni a kezdő állapotokat. A *Parser* konstruktor paramétere egy lista, amelyben felsorolásra kerül minden abban található *state* kulcsszóval kezdődő függvény.

Ezek az állítások tartalmazhatják egy fejléc kibontását (*.extract())*, egy másik *state*-re való ugrást (*transition*), vagy pedig egy elágazást (*transition select*). Egy P4 program futásakor az elágazás során egy bizonyos értéket vizsgálva dől el, hogy melyik végrehajtással folytatódik a *parser*. A hiba észlelés számításakor viszont ez az érték számunkra nem releváns, az elemző az összes lehetséges végkimenetelt vizsgálja, és a kezdő állapotok közé felveszi.

##### Control elemző függvények

A *control* kulcsszóval kezdődő függvények elemzésére használt függvények.

Ezek a benne található *action*, *table* és *apply* kulcsszavak mentén ismerik fel a definiált akciókat, táblákat, valamint az alkalmazandó programot.

Ezek a *ParserAction*, *ParserTable* és *Apply* konstruktorokkal jönnek létre.

Az akcióknak a neve, és a hozzá tartozó szekvenciába rendezett értékadások és függvényhívások kerülnek rögzítésre.

A tábláknál fontos információ a kulcsok, valamint a hozzájuk rendelt akciók nevei.

Az *apply* kulcsszóval jelölt függvényben, mely a kontrollfüggvény törzse, általában elágazások(*if-else*), akció- és táblahívások, valamint *.emit(fejléc)* függvényhívások szerepelnek. Ezek is elmentésre kerülnek, egy listában.

##### Kifejezéseket elemző függvények

Ezek az aritmetikai és logikai kifejezések, valamint a függvényhívások elemzéséhez szükséges függvények.

Mindegyik a hozzájuk tartozó, nevükkel megegyező adattípusokat ismeri fel, és bontja fel a megfelelő változóra és operátorra. Ebben az esetben a függvényhívások operátorként vannak definiálva.

Az operátorok esetében az is rögzítve van, hogy egy kifejezés elején, közepén vagy végén található-e meg. A jól ismert összeadás jel (+) például a kifejezés belsejében, de a függvényhívások, mint a fejléc validitásának beállítására használt .setValid(), a kifejezések végén fog előfordulni. Ez az információ a mintaillesztést segíti.

### Preparation.hs

Miután a szintaktikai elemző a szükséges információkat kiszedte a programból, átadja ezt a *Statement* listát az előkészítő lépésnek.

Ez rekurzív módon halad végig a listán, és minden egyes elemet megfelelő alakra alakít át. A fejlécekből és azok mezőiből hozza létre a kezdő- és végállapotokat, az *Apply*-ból és az ahhoz tartozó akciókból és táblákból pedig az elemzendő programot, amelyen végighaladva fognak a kezdőállapotok módosulni.

Továbbá itt kerül átalakításra a felhasználói felülettől kapott mellékfeltételek szövege egy megfelelő adattípussá.

Ennek a résznek az a célja, hogy egy köztes nyelvre hozzuk a megkapott programunkat, amelyben jelen vannak a P4 specifikus elemek, de mégis könnyebb a későbbiekben elemezni.

#### Bevezetett adattípusok

Az előkészítő folyamat az elemzőben megtalálható adattípusokat konvertálja át az ellenőrző állományban lévő adattípusokra. Ezek a dokumentációban a saját alcímük alatt vannak bővebben taglalva.

A konstruktorok paraméterei általában *string*-ek, vagy további adattípusok *string* paraméterrel. Ebből kifolyólag az átírás nem okoz bonyolult konverziókat.

A Parser és a Verification kódállományokban lévő adattípusok konstruktorainak nevei nagyon hasonlóak, ezzel is könnyítve az átalakításokat.

#### Függvények

##### mainConversion

Az a rekurzív függvény, amely paraméterbeli listának az elemeihez a megfelelő kisebb átalakító függvényt rendeli. Ezt a konstruktor alapján dönti el.

Paraméterként megkapja az elemző által készített *Statement* listát, egy üres környezet listát a kezdeti állapotoknak, egy üres környezetet a végállapotoknak, valamint három üres *Program* listát, amelyben rendre az akciókat, a táblákat és az *apply*-ból kinyert programokat fogja tárolni.

Ha az átalakítás során bármely résznél hiba merülne fel, akkor a rekurzió leáll, és a paramétereknek megfelelő *error* konstruktorokat adja vissza. Ha nincs hiba, a rekurzió végeztél végleges alakra hozza a környezeteket.

A kezdőállapotok esetében ez azt jelenti, hogy az alap *Environment* típusról *IdEnvironment* adattípusra alakítja át, amely az előzőt egészíti ki egy azonosítóval, ami a megjelenítésnél létrehozott gráfnál lesz fontos, valamint egy *EnvironmentType* típusú adattaggal, aminek értéke lehet Match, NoMatch és Stuck. Ezek rendre azt jelelik, hogy a környezet sikeresen elért a számítás végére és megegyezik a végállapotok valamelyikével, vagy elért a végére, de egyik végállapottal sem azonos, vagy a számítás során elakadt.

A végállapotnál kettő környezet kerül létrehozásra. Az egyik a program *deparser* részében megtalálható *.emit(fejléc)* függvények által, a másik a számítás végén kerül hozzáadásra. Ez egy alapértelmezett végállapot, amelyben a *drop* értéke *Valid*, így minden más fejléc és mező *Undefined* lesz.

##### Fejléceket átalakító függvények

Ha a *Statement* listában a *mainConversion* függvény *ParserHeader* vagy *ParserStruct* konstruktorokat talál, melyek a fejlécek nevét, valamint a hozzájuk tartozó mezők neveinek listáját tartalmazzák, akkor a kezdő- és végállapotokat tároló paramétereit a *headerConversion* függvénynek adja át.

Ez fogja a fejlécet a megfelelő alakra hozva beépíteni a környezetekbe. A fejléc nevéhez rendel egy validitást, amely a kezdőkörnyezetek esetében *Invalid*, a végállapotok környezeteiben pedig *Undefined*, majd ugyenezt végrehajta minden mezőn is, a fejléccel azonos validitással a *fieldsConversion* függvény segítségével.

##### parserConversion

*Parser* konstruktor esetén a *mainConversion* a kezdőállapotokat tartalmazó paraméterével hívja meg a *parserConversion* függvényt.

Ez a listaelem az elemzés során egy *state* elemeket tartalmazó listává lett átalakítva, így a feldolgozáskor ezen kisebb részeket haladunk végig.

Első lépésként megkeresi a *start* elnevezésű elemet, amely egy P4 program futtatásakor minden esetben legelőször fut le. Ha nem talál ilyet, akkor egy hibát jelző adattípussal tér vissza, amely a későbbiek során egy hibaüzenetté lesz átalakítva, így továbbítva a felhasználó felé.

Ha sikeresen megtalálta a kezdő elemet, akkor megkezdődik az elemek feldolgozása a *stateConversion* függvény segítségével.

Egy *state* utasításai közül számunkra három lényeges. Ezek a *transition*, amely a meghatározza, hogy mely elemen folytatódik a feldolgozás, a *transition select*, amely az előzőhez hasonló, csak feltételes elágazással, valamint az *.extract()* függvény, amely egy fejlécre meghívva kibontja azt. Ennek reprezentálása a fejléc és annak mezőihez tartozó validitások megváltoztatása *Valid*-ra.

A *stateConversion* minden esetben megvizsgálja, hogy a *state* tartalmaz-e kicsomagoló függvényt. Ha igen, akkor azt alkalmazza a környezetre, ha nem, akkor a környezet változatlanul marad.

A feldolgozás folyatása pedig a *transition* és *transition select* utasítások szerint halad. Ez a kettő egyszerre nem szerepel, tehát vagy egyértelműen halad a feldolgozás, vagy pedig elágazik. Ha egyik sem található meg az elemben, akkor ott a rekurzió megáll, miután az *.extract()* függvények alkalmazásra kerültek, ha voltak.

A *transition* esetében a kezdő környezetek száma változatlan marad, a *stateConversion* újrahívása a megadott nevű state elemmel folytatódik.

A *transition select* feltételeket és *state* név párokat tartalmaz. A feldolgozás során a feltételekkel nem foglalkozik, az elágazás minden lehetséges ágán végighalad, így a környezetek mennyisége ezek számával arányosan növekszik.

Miután a függvény a fent leírt módon végighaladt a *Parser* összes elemén, a kezdőállapotokat leíró paraméter az összes lehetséges környezetet tartalmazni fogja, amellyel a P4 program lefutásra kerülhet. Az ellenőrzés során a program az ebben lévő környezeteken fogja a számításokat elvégezni.

##### controlConversion

Ha a *Control* a következő konstruktor a *mainConversion* futása során, akkor az erre megírt *controlConversion* függvényt fogja meghívni. Ennek a kezdő – és végállapotokat, az akciókat, táblákat, és a programot tartalmazó paramétereket fogja átadni.

A feldolgozás ebben az esetben is a lista elemein fog végighaladni.

* A *ParserAction* esetében az *actionConversion* kerül meghívásra. Egy akciónak van neve, valamint utasításai, ezek a megfelelő átírás után szekvenciálisan vannak eltárolva. Ezután az átalakítás után az akciókat tartalmazó paraméterlista végére fűzzük, mint új akciót.
* A *tableConversion* függvényt hívjuk meg, ha *ParserTable* típusú a következő elem. Az átírás során a tábla neve és kulcsai mellett tárolásra kerülnek a táblához tartozó akciók. Ezeket az előzőleg említett akció listából másoljuk át a tábla paramétereihez.
* Az *Apply* tartalmazza a programot, ha ilyen konstruktort talál a *controlConversion* függvény, akkor az *applyConversion* függvényt fogja meghívni. Ezáltal lesz majd az a program létrehozva, amin végighaladva az ellenőrzés végrehajtódik.
* Az Apply paraméterében megkapott programot alakítja át, majd adja vissza úgy, hogy minden akcióhívás alkalmával beilleszti az akciót, a táblahívások során a táblát, annak akcióival együtt. Az elágazások és szekvenciák a programnak megfelelően megmaradnak.
* Ha a kontrollfüggvény egy *deparser*, akkor ott az *Apply* konstruktor paraméterében *.emit()* függvényhívások szerepelnek. Ezért, minden esetben, amikor *Apply* konstruktort talál, a *controlConversion* egy másik függvényt is meghív a végállapotokat tartalmazó paraméterével.
* Ez az *emitConversion* függvény, amely minden *.emit()* hívásban lévő fejléc esetében *Valid*-ra állítja annak minden értékét. Így kerül létrehozásra, a *drop* *Valid* környezet mellett, a másik végállapot. Utóbbi tehát az átalakítás közben, míg előbbi majd csak az átalakítások végén jön létre.

##### sideConditionConversion

Fontos még a mellékfeltételek átalakítása, melyet a program felhasználói felület részétől kapunk meg, számok sorozataként.

A sorozat minden eleme egy programfüggvényhez tartózó mellékfeltételt jelöl, míg az értéke azt, hogy van-e ellenőrzés, és ha igen, akkor *Valid* vagy *Invalid* érték esetén teljesül a feltétel. Egy adott programfüggvényhez több mellékfeltétel is tartozik, ez a sorozatban a számokat elválasztó & szimbólummal van elválasztva.

A programfüggvények, amelyekhez rendeljük a mellékfeltételeket rendre az elágazás, tábla, értékadás, fejléc és drop. Ezek az ellenőrzés során fontosak, így a Verification.hs részben vannak bővebben kifejtve.

##### Adattípusokat szöveggé alakító függvények

A még ehhez a részhez tartozó, de kisebb egység a szöveggé alakító függvények. Ezek a Haskellben definiált és használt adattípusokat konvertálja szöveggé, hogy az eredményeket vissza lehessen küldeni a felület rétegnek.

### Verification.hs

Miután az előkészítő folyamat sikeresen a köztes nyelvre hozta a programot, négy fontos adat áll rendelkezésünkre az ellenőrzéshez. A végállapotok, kezdőállapotok, a program és a mellékfeltételek.

A hiba észlelés folyamata a program mentén zajlik. Rekurzív hívásokkal mindig az pillanatnyilag illeszkedő szabályt választjuk ki, és alkalmazzuk azt a környezetekre, ha a mellékfeltételek teljesülnek.

#### Bevezetett adattípusok

##### Validity

A környezetekben szereplő fejlécek és azokhoz tartozó mezők érvényességének jelöléséhez létrehozott adattípus. Értéke lehet *Valid*, ha érvényes, *Invalid*, ha nem az, *Undefined*, ha nem meghatározott a fejléc vagy mező és *None*, ami a mellékfeltételekhez szükséges, ha nincs ellenőrzés. Az egyenlőség vizsgálat során az *Undefined* validitás minden mással egyenlő.

##### Field

A fejlécekhez tartozó mezők típusa. Egy név és validitás párból áll.

##### Header

A fejléc típusa. A nevén és a validitásán kívül a hozzá tartozó mezők listájából épül fel.

##### Environment

A környezet adattípusa lényegében fejlécek listája. Konstruktora az *Env*. Minden esetben tartalmaz egy *drop* nevű fejlécet, amelynek mező listái üresek.

Az ellenőrzés során csak a végállapotok lesznek ilyen típusúak, mivel azok számossága mindig kettő lesz. A *deparser* által kiszámított, valamint a valid *drop* érték esetén meghatározott környezetek. A kezdő, valamint azokból kiszámított környezetek azonosítóval vannak ellátva.

Másik konstruktora az *EnvError*, mely hiba észlelése esetén egyértelműsítheti, hogy a számítás során a környezetek alakítása során történt hiba. Ha az átalakítások vagy átírások során *EnvError* a visszakapott eredmény, az éppen zajló folyamat megáll, és ezt küldi vissza a felhasználói felületnek.

##### EnvType

Az *IdEnvironment* kiegészítéséhez szükséges típus, mellyel számontartom, hogy az adott környezet milyen állapotban van. Ha a környezet a számítás során mindig kielégítette a mellékfeltételeket és megegyezik valamelyik végállapottal, akkor az értéke *Match*. Ha az előzőhöz hasonló, de mégsem azonos a végállapotok egyikével sem, akkor *NoMatch*. Ha pedig a számítás során nem teljesült rá a mellékfeltételek egyike, akkor elakadt, vagyis az értéke *Stuck*.

##### IdEnvironment

Az *IdEnvironment* a környezet típus olyan kiegészítése, amely a fejléc lista mellett tartalmaz egy egyedi azonosítót és egy típust, amely a környezet állapotát jelöli.

Az azonosító kezdetben csak egy szám, a kezdőkörnyezeteket sorszámozom. Ezután minden lépésben, amikor alkalmazva van egy programfüggvény, akkor annak nevét hozzáfűzzük az azonosító végére. Így a számítás végére minden kiszámolt környezet azonosítójából kinyerhető, hogy milyen lépések során jutottunk el abba az állapotba. Ez a felhasználói felület során a gráf megépítésénél tölt be fontos szerepet.

##### Program

A köztes nyelv reprezentációja. Konstruktorai egy-egy programstruktúrát határoznak meg, a számításhoz szükséges módon leegyszerűsítve.

* Az *EmptyProg* a köztes nyelvre való átírás során hasznos. Jelzi, ha a felhasználó olyan programot adott, amely vagy teljesen üres, vagy nincs olyan része, amely alkalmas lenne számítások végzésére.
* A *ProgError* az *EnvError* kontruktorhoz hasonlóan hibajelzésre alkalmas, mely nagyobb szintaktikai hibák vagy a résznyelven kívül eső programrészekre figyelmeztet, és szintén leállítja a folyamatokat. Feltételezve van, hogy a beadott program egy leforduló, helyes P4 program, így a hibaészlelés nem a teljes P4 nyelvre kiterjedő.
* A *Skip* a standard ugrás programstruktúra, amely során nem történik semmilyen változtatás. *If-else* struktúra üres *else* ágában fordul elő a leggyakrabban, de a szekvencia második részeként is sokszor megjelenik.
* A *Seq* a szekvencia konstruktora. Ennek paramétere két *Program*, így egymás után bármilyen programstruktúra előfordulhat.
* Az elágazást az *If* konstruktorral van reprezentálva. Paraméterei az feltételben szereplő változók, és két *Program* amik az igaz illetve a hamis ágban vannak.
* A P4 specifikus tábla konstruktora a *Table*. A tábla neve, a kulcsainak valamint az akcióinak a listája a hozzá tartozó paraméterek.
* Az *Action* az akció konstruktora. Ennek paramétere az akció neve, valamint a szekvenciálisan egymásba ágyazott utasítások. Ezek a következő három valamelyike lehetnek.
* Az értékadást *Assignment* konstruktorral reprezentáljuk. Az egyenlőség bal oldalán szereplő változó az első paraméterei, és minden más változó, amely a jobb oldalon szerepel a második paraméterként meghatározott listában van tárolva. Ha az értékadás során a jobb oldalon konstans érték szerepel, akkor ez a lista üres.
* A *Drop* konstruktor a *mark\_to\_drop()* függvényhívást jelöli. Nincs paramétere, az egyértelmű *drop* fejléc módosítása miatt.
* A fejlécekre meghívható *.setValid()* és *.setInvalid()* függvények reprezentálása a  *SetHeaderValidity* konstruktor. Paraméterei a fejléc neve, valamint a megfelelő validitás érték.

##### SideCondition

A felhasználók szigoríthatják az ellenőrzéseket különböző ellenőrzések megadásával. A programfüggvényekhez több feltétel is rendelhető. Az adott programstruktúrához rendelt feltételek mindegyikének helyesnek kell lennie ahhoz, hogy a módosítás végrehajtásra kerüljön. Ha hamis, akkor az adott környezet nem módosul. Minden környezetre külön történik meg a vizsgálat.

A mellékfeltételek reprezentációja egy validitás lista ötös. Ezek *None*, *Valid* és *Invalid* értékeket vehetnek fel. Ezek a feltételek konjunkcióban állnak egymással, és ha bármelyik None értékkel rendelkezik, akkor arra a feltételre nem történik ellenőrzés, és alapértelmezetten igaz lesz. Ha Valid vagy Invalid az értékük, akkor ilyen validitással kell rendelkeznie annak az elemnek, amelyre a vizsgálat megtörténik, hogy a feltétel igaz legyen.

A listák rendre a következők.

* Az első kételemű lista az elágazás mellékfeltételeit tartalmazza. Az egyik az elágazás feltételében szereplő mezők validitását vizsgálja, míg a másik a fejlécekét.
* A második szintén egy kételemű lista a táblák mellékfeltételeihez. Ezek a kulcsokra alkalmazhatóak, az első a mezők, a második a fejlécek validitásának ellenőrzését teszi lehetővé.
* A harmadik egy négyelemű validitás lista, amely az értékadás mellékfeltételeit tartalmazza. Ellenőrzés történhet az értékadás bal oldalán lévő mezőre és fejlécre, valamint a jobb oldalán álló mezőkre és fejlécekre.
* A negyedik lista fejlécek állítására vonatkozik. Itt két mellékfeltétel van, amelyek az adott fejléc, valamint annak összes mezőjének validitását ellenőrzik.
* A *drop* mellékfeltételei az ötödik háromelemű listában van. A szabályok alkalmazása előtt vizsgálásra kerülhet a *drop*, az adott környezetben szereplő összes fejléc, valamint az összes mező validitásának értéke.

##### Rule

A *Rule* egy lambda függvény, amelynek mintaillesztésével választjuk ki a megfelelő programfüggvényt a program pillanatnyi állapotához.

Ez mindig az adott konstruktor alapján kerül kiválasztásra. A szabályokból tehát több van, amely egy *Rules* listában van tárolva.

A lambda felépítése a következő. Paraméterei a számítás során tárolt környezetek listája (*IdEnvironment*), a program (*Program*), a mellékfeltételek (*SideCondition*) valamint egy szám, amely a környezet azonosítójának egyediségének megtartásához szükséges.

#### Függvények

##### verifyP4

Az ellenőrzést lefuttató függvény. Paraméterei a szabály típussal azonosak. Az összetett programstruktúrák függvényei a lefutásuk után ezt a függvény hívják meg újból, így az ellenőrzés rekurzívan folyik.

Kezdetben a kezdőállapotokkal van meghívva, a futása végén pedig a kiszámított végállapotokat kapjuk meg.

##### fittingRule

A megfelelő szabály kiválasztásához használt függvény. A Program adott konstruktorával illeszt mintát a *Rules* listán, és az illeszkedő szabályt adja vissza.

Ezt a szabályt fogja a *verifyP4* alkalmazni, ami annyit tesz, hogy meghívja a megfelelő programfüggvényt a szükséges paraméterek átadásával.

##### Programfüggvények

Minden *Program* konstruktorhoz tartozik egy programfüggvény.

* A *prFunc\_Skip* a megkapott környezetlistát változatlanul visszaadja.
* A *prFunc\_Drop* a környezetlista mellett megkapja a *Drop* konstruktorhoz tartozó mellékfeltételeket, valamint a számot, amely az azonosítókhoz szükséges. A függvény egyesével halad végig a környezeteken, és mindegyiken ellenőrzi a mellékfeltételeket és ettől függően módosítja azt. Ha a feltételek mindegyike igaz, akkor a *drop* értékét *Valid*-ra állítja a környezetben, minden mást változatlanul hagy. Ha nem, akkor a környezet típusát *Stuck*-ra állítja. Mindkét esetben az azonosító végére fűzi a „drop” szöveget, valamint a paraméterként megkapott számot. Ha az összes környezettel végzett, akkor visszaadja a módosított listát.
* A *prFunc\_SetHeaderValidity* paraméterei a környezetlista, a fejléc neve, a kívánt validitás, valamint a mellékfeltételek és az azonosítóhoz szükséges szám. Minden környezet esetében ellenőrzi a mellékfeltételeket. Ha igaz, akkor megkeresi az adott környezetben a fejlécet, és a megadott validitásra állítja át az értékét. Ha hamis, akkor a környezet típusa *Stuck* lesz. Az azonosító végére a „setHeader” szöveget és a paraméterként megkapott számot fűzi. A folyamat végén visszaadja a módosított környezeteket tartalmazó listát.
* A *prFunc\_Assignment* megkapja a környezetlistát, az értékadás bal és jobb oldalán álló változókat, a mellékfeltételeket és az azonosítóhoz szükséges számot. A függvény az összes környezet esetében ellenőrzi a mellékfeltételeket, és ha mind igaz, akkor megkeresi a bal oldalon szereplő változót a környezetben a fejlécek és a mezők között, és átállítja az értékét *Valid*-ra. Ha hamis, akkor átállítja a környezet típusát *Stuck*-ra. Az azonosító végére fűzi az „assignment” szöveget, valamint a bal oldalon szereplő változót, és a számot. A visszatérési értéke a módosított környezetlista.
* A *prFunc\_Action* az akciókhoz tartozó mellékfeltételek mellet megkapja a környezetlistát, az azonosítóhoz szükséges számot, az akció nevét, és az ahhoz tartozó szekvenciálisan egymásba ágyazott utasításokat. Az összes környezet azonosítójához hozzáfűzi az „action” szöveget, a számot, és az akció nevét. Az azonosító számot növeli eggyel, és ezzel, valamint az új környezetlistával hívja meg a *verifyP4* függvényt.
* A *prFunc\_If* az elágazáshoz tartozó programfüggvény. Az elemzés csak a két ágú *if-else* típusú elágazással foglalkozik. A függvény paraméterei a környezetlista, a mellékfeltételek, az azonosítóhoz szükséges szám, az elágazás feltételei és a két *Program* típusú paramétere. A futása során először minden környezet esetében ellenőrzi a mellékfeltételeket. Azoknak a típusát, amikre nem teljesülnek, *Stuck*-ra változtatja. Ezután a rész után a függvény két külön részre szedi a környezeteket, az egyik listában lesznek a megakadt környezetek, a másikban az összes többi. Ezután a nem *Stuck* típusú környezetek számát duplázza úgy, hogy külön-külön meghívja verifyP4 függvényt az elágazás igaz és hamis ágához tartozó *Program* paraméterrel is. A környezetek ekkor megkapják az új azonosítójukat is, mely az eredetiből, valamint az igaz ág esetében az „if”, a hamis ág esetében az „else” szövegből és az paraméterben szereplő számból áll. Az ezektől visszakapott két listát végül egymással, és a *Stuck* típusú környezeteket tartalmazó listával fűzi össze, és ez a visszatérési értéke.
* A *prFunc\_Seq* paraméterei a környezetlista, a mellékfeltételek, valamint az azonosítóhoz szükséges szám. Ezek nem kerülnek módosításra, a függvény csak a szekvenciális végrehajtás szimulálja, valamint kiszűri a *Stuck* típussal rendelkező elemeket, és azok nélkül hívja a *verifyP4* függvényt.
* A *prFunc\_Table* programfüggvény a tábla módosításait hajtja végre. Ez egy többágú elágazás a megadott akciók mentén. Paraméterei a környezetlista, a mellékfeltételek, az azonosítóhoz szükséges szám, valamint a tábla neve, kulcsai és akciókat tartalmazó listája. Először a mellékfeltételek kerülnek ellenőrzésre, majd a függvény kiszűri azokat a környezeteket, amelyek elakadtak, és eltárolja őket egy listában. Az el nem akadt környezetek szintén egy listába kerülnek. A visszatérési értéke egy környezetlista lesz, amely három listából fog állni. Az első lista az elakadt környezetek listája. A második lista úgy jön létre, hogy a függvény az akció listában lévő legelső akcióval hívja meg a verifyP4 függvényt, miközben a környezetek azonosítóit kiegészíti a „table” szöveggel, a tábla nevével és az azonosító számmal. A harmadik listában a függvény önmagát hívja meg rekurzívan az el nem akadt környezetekkel, és az akció lista további elemeivel. A rekurzió akkor fejeződik be, ha ez a folyamat minden akció esetében megtörtént. A környezetlista elemszáma így növekszik, azok a környezetek, amelyekre igazak a tábla mellékfeltételei annyiszor jelennek meg az állapotlistában, ahány akció a táblához van rendelve. Minden újabb környezetre csak az adott akció lesz végrehajtva, így szimulálva az elágazást.

##### Mellékfeltételeket ellenőrző függvények

Minden programfüggvény a saját mellékfeltételeket ellenőrző függvényét hívja meg a fentebb leírt módokon. Ezek egyszerre mindig csak egy környezettel dolgoznak.

A függvények az összes feltételt külön ellenőrzik, majd az eredményeket egy nagy konjunktív kifejezésként adják vissza, amely egyetlen logikai értéket jelent.

##### compareCalculatedWithFinal

A számítás végeztével a kiszámított környezetek vagy *Stuck* vagy *NoMatch* típussal rendelkeznek. A *compareCalculatedWithFinal* függvény a végső összehasonlítást végzi, amely során minden kiszámított végállapotot összehasonlít a programból kinyert végállapotokkal, de csak akkor, ha az adott környezet nem *Stuck* típusú, mert akkor nincs értelme az összevetésnek. A függvény tehát visszaadja a kiszámított állapotokat úgy, hogy minden *NoMatch* típusú környezetet összehasonlít mindkét adott végállapottal, és ha bármelyikkel is egyezik, akkor a típusát *Match*-ra fogja állítani, egyébként *NoMatch* marad.

### Calculation.hs

Ez az állomány azt a függvényt tartalmazza, amelyet a felhasználói felület modellje fog hívni. Az összes eddigi .hs kiterjesztésű fájlt importálja, így a .dll kiterjesztéssé konvertálás során minden függvényt tartalmazni fog a könyvtár.

A konvertálás parancsa a *ghc --make -shared calculation.hs*. Az így kapott .dll fájlt a felhasználói felület projektjében elhelyezve könnyen hozzáfér a dinamikus könyvtárhoz.

##### cCalculate

A függvény, amelyet a modell hív. A paraméterek a P4 program és a megadott mellékfeltételek egy „&” szimbólummal elválasztott számsorozata.

Az így kapott két szöveget *CWString* típussal jelöljük, ez olyan C típusú nyelvekben használt szöveg, amely karaktertömbként van reprezentálva, és null elemmel végződik.

A függvény első lépésként ezt a szöveget konvertálja át a Haskellben használt *String*-re. Ezután a *helpCalculate*-nek átadja őket, majd az ettől visszakapott eredményt fogja átalakítani egy *CWString*-re, amelyet visszaküld a felhasználói felületnek.

##### helpCalculate

A *cCalculate* függvénytől kapott programot és mellékfeltételeket felhasználva az ellenőrző folyamat megfelelő sorrendjében hívja meg minden .hs állomány főbb függvényét.

Először a *Parser.hs parseString* függvényét hívja meg a megkapott programmal, amely elvégzi az elemző lépéseket.

Ezután a visszakapott *Statement* listával meghívja a *Preparation.hs mainConversion*, a mellékfeltételekkel pedig a *sideConditionConversion* függvényét. A köztes nyelvre hozott programot, a környezeteket kapja meg az előbbitől, a validitás listákkal reprezentált feltételeket az utóbbitól.

Itt történnek meg az ellenőrzések arra az esetre, ha a program vagy a környezetek valamilyen hibára futottak volna. Ebben az esetben az ellenőrző rész már nem kerül lefutásra, hanem hibaüzenetet küld vissza a felhasználói felületnek.

Ha nincs hiba, akkor a *NOERROR* üzenetet fogja továbbítani az eredményekkel együtt. Ez egyetlen szöveggé van összefűzve a következő módon.

Négy nagyobb részből tevődik össze, ezek az „&” szimbólummal vannak elválasztva, és rendre a hibaüzenet, a kiszámított és összehasonlított végállapotok, a programból kinyert végállapotok és a kezdőállapotok.

Ha hiba történik, akkor a hibaüzenet után szerepel az „&” szimbólum, viszont a további részek nem szerepelnek.

Az ellenőrző rész a *verifyP4* hívásával zajlik, amelynek a köztes nyelvre hozott programot és a mellékfeltételeket, valamint a kezdőkörnyezeteket adjuk át.

A kiszámítás után még megtörténik a *compareCalculatedWithFinal* hívása, amellyel véglegesítjük a kiszámított állapotokat.

Ezután visszaadja a fent leírt módon összefűzött szöveget a *cCalculate* függvénynek, amely visszaküldi az eredményt a felhasználói felületnek.

## 3.4. Felhasználói felület

A felhasználó felület *C#* nyelven íródott *Visual Studio* fejlesztőkörnyezetben.

Az implementálásához a *Windows Presentation Foundation* osztálykönyvtár lett felhasználva. A felépítése az egyetemen megismert *Model-View-ViewModel* szerkezetű. A *Haskell* részben megírt ellenőrző algoritmus egy .dll fájlként kerül a modellhez, de annak szerkezetileg nem része, az csak függvényét használja.

### Model

Az üzleti logika rész feladata a .dll fájllal való kommunikáció, valamint az attól kapott eredmények megfelelő alakra hozása.

#### IdEnvironment.cs

Az *IdEnvironment* osztályt tartalmazó állomány. Minden az ellenőrző résztől kapott állapot ilyen objektumként van létrehozva. Az osztály az állapotok reprezentálása, melynek három mezője van.

A *LeafEnv* egy szöveg, amely magát az állapotot tartalmazza, elnevezése arra is utal, hogy a gráf kirajzolása során a környezetek a leveleken fognak elhelyezkedni.

Az *EnvType* a környezet típusát jelöli, amely lehet *NoMatch*, *Match* és *Stuck*. A gráf levelei annak megfelelően vannak színezve, hogy ez melyik az előbbi három közül, rendre sárgára, zöldre és pirosra.

Az *EnvId* tartalmazza az azonosítót, melyet lebontva megkapjuk, hogy melyik kezdőállapotból milyen lépések után kaptuk meg az adott környezetet. A gráf felépítésénél fontos.

#### HaskellCalculation.cs

Ez a fájl a HaskellCalculation osztályt tartalmazza.

Ennek három *DllImport* attribútummal rendelkező függvénye van, a *Calculation.dll* inicializálásához, a bezárásához, valamint a *cCalculate* függvény meghívásához. Az elsőt az osztály konstruktora során hívjuk meg, a másodikat az objektum megszűnésekor.

A harmadikat a publikus *HsCalculate* függvényen keresztül érjük el. Ennek paraméterei a program és a mellékfeltételek.

#### Események

A *Model* és a *ViewModel* közötti kommunikáció eseményeken keresztül zajlik. A modell irányából kétféle információ érkezhet. Vagy a számítás végeztével adja oda az eredményeket, vagy valamilyen hiba esetén hibaüzenetet küld a *ViewModel* felé.

##### CalculationEventArgs.cs

Ennek az eseménynek három mezője van. Ezek tárolják az eredményből kiszedett információk közül a kiszámított végállapotokat, a programból kinyert végállapotokat, és a kezdőállapotokat. Ezeket *IdEnvironment* objektumokat tartalmazó listaként adja a *ViewModel*-nek.

##### ErrorEventArgs.cs

Az eredményben megkapott hibaüzenetet ez az esemény fogja a ViewModel részére átadni. Ennek tehát csak egy szöveg mezője van, amely magát az üzenetet tartalmazza.

#### VerificationModel.cs

A modell törzse a *VerificationModel*. Ez az az osztály, amely a *Calculation.dll* könyvtárral folytatja a kommunikációt.

##### Mezők

Az osztályban használt értékek a C# nyelvben használt publikus *getter/setter* típusú mezőkben kerülnek tárolásra. Ezekhez tartozik egy privát változó is, melyeket a program többi rétege nem ér el.

* Az *InputString* változóban kerül tárolásra a felületen a felhasználó által beadott program. Ez egy szöveg típusú mező. Értéke a kalkuláció elindításának pillanatában kerül a modell résznek átadásra, így ekkor kapja meg az értékét.
* Az *ErrorString* szöveg az ellenőrzés során fellépett hibaüzenetet tárolja el. Ennek értékét küldi a *ViewModel* számára az erre megírt *ErrorEventArgs*.
* A *hscalculation* a *HaskellCalculation* osztály objektuma. Ennek segítségével hívható meg a Haskellben megírt ellenőrzés.
* Az *Environments*, az *InitEnvs* és a *FinalEnvs* mind egy-egy *IdEnvironment* típusú objektumokat tartalmazó lista. Ezek a kalkulációtól megkapott kiszámított környezeteket, a programból kinyert kezdő-, valamint végállapotokat tartalmazzák.

##### Események

Az *Error* és a *CalculationDone* az *ErrorEventArgs* és a *CalculationEventArgs* osztályok objektumai. Az előbbi az *ErrorString* értékét, míg utóbbi az *Environments*, az *InitEnvs* és a *FinalEnvs* listákat továbbítja.

##### Függvények

Az modell feladata az ellenőrzés elindítása, majd az eredmény feldolgozása. Függvényei ennek megfelelően a következőek.

* Az osztály konstruktora paraméter nélküli, és a mezőknek ad megfelelő kezdőértéket.
* A *Calculate* a *ViewModel* által hívott függvény, melynek paramétere a *View* rétegtől megkapott bemenő program és a mellékfeltételek szövegként. Első lépésként az osztály minden mezőjét alaphelyzetbe állítja. Ezután az InputString mezőjének értékül adja a megkapott P4 programot, mint szöveget. Ez persze ellenőrzésre kerül, ha üres szöveg lenne, akkor az ErrorString ennek megfelelő hibaüzenetet fog tartalmazni. Ha az input nem üres, az ellenőrzés megtörténik a *hscalculation* objektum *cCalculate* függvényével, a megfelelő két paramétert átadva. A számítás végeztével ellenőrzésre kerül az az által meghatározott hibaüzenet. Ha hiba történt, akkor az *ErrorString* ennek megfelelő hibaüzenetet fog tartalmazni. Ha nem, akkor az eredmény feldolgozása következik. Ezt az következő három függvény fogja megtenni. Miután kész a feldolgozás, a két eseményen keresztül megkapja a *ViewModel* az eredményeket.
* Miután a *Calculate* három részre darabolta az eredmény szöveget, azokat a *processEnvs*, a *processFinalEnvs* és a *processInitEnvs* függvényeknek adja oda. Ezek a szövegen végighaladva készítenek új *IdEnvironment* objektumokat, és azokat a nekik megfelelő lista (*Environments*, *FinalEnvs*, *InitEnvs*) végére fűzik.

### ViewModel

A *ViewModel* a *Model* és a *View* rétegek között szerepel. Feladata a felülettel kapcsolatos adatok tárolása és kezelése, az ezekhez tartozó metódusok és parancsok implementálása. Az ebben előforduló listák mind *ObservableCollection* típusúak a megjeleníthetőség érdekében.

#### Condition.cs

A mellékfeltételek reprezentálásához létrehozott osztály. Ennek segítségével a felületen legördülő menüben szereplő, valamint az ezekben kiválasztott értékek kerülnek tárolásra.

A *Condition* ősosztályban szerepel egy lista, a *CondsCheck*, amely rendre a „Nincs”, „Valid” és „Invalid” szövegeket tartalmazza. Ez a lista az összes legördülő menühöz hozzá van rendelve, a felhasználó ezen értékeket állíthatja be egy-egy mellékfeltételhez.

Az ősosztályból származtatva, minden programstruktúrához tartozik egy osztály. Ezek a programstruktúrák az *if-else* elágazás, a tábla, értékadás, fejléc módosítás és drop. Az osztályok az ehhez tartozó mellékfeltételeket egy számként reprezentálják. Konstruktoruk ezeket alapértelmezetten 0-ra állítja, mely a „Nincs” szöveggel társítja azt, amely azt jelenti, hogy ilyen mellékfeltétel ellenőrzés nem lesz elvégezve a számítás közben. Az 1-es ennek megfelelően a „Valid”, míg a 2-es az „Invalid” szövegeknek van megfeleltetve, és azt jelentik, hogy a megadott validitás szerint történjen ellenőrzés a mellékfeltétel szerint.

Az osztályhoz továbbá tartozik egy esemény is, mely a *VerificationViewModel* számára küld jelzést, ha a legördülő menük valamelyike használva volt.

#### DelegateCommand.cs

A ViewModel rétegben lévő parancsokat ebből az osztályból származtatjuk. Mivel az események összekötnék a felületet a modellel ezek parancsokkal vannak helyettesítve.

Ezek a felületen szereplő vezérlők *Command* tulajdonságához vannak kapcsolva. A végrehajtás során a paraméterben megkapott tevékenységet fogja elvégezni.

Mivel több ilyen parancs van, ezért volt érdemes egy külön ősosztályban elhelyezni az ehhez szükséges mezőket és metódusokat. A végrehajtandó tevékenység egy lambda kifejezésként fog megjelenni a konstruktor paraméterében.

Ez az osztály a tanulmányaim során egy tárgy keretein belül több alkalmazásomhoz is fel lett használva.

#### ViewModelBase.cs

A *ViewModel* változásjelzéssel való kiegészítéséhez egy ősosztály lett létrehozva, mely a *ViewModelBase* osztály. Ez az *INotifyPropertyChanged* osztályból van származtatva. Ennek segítségével egyszerűbben jelezhetjük a View felé, ha valamelyik megjelenített érték megváltozott.

#### P4Graph.cs

Az eredmények grafikus megjelenítésének módját gráffal oldottam meg.

Ehhez egy már létező könyvtárat alkalmaztam, amit *NuGet* csomag formájában adtam hozzá a projekthez. Ennek neve *GraphSharp*. Ez egy olyan könyvtár, amely egy másik, különböző gráfok reprezentálásának és algoritmusainak gyűjteményén alapszik. Ez a *QuickGraph*.

Ezen könyvtáraknak csak nagyon kevés eleme lett felhasználva, mivel a megjelenítés csak egy fa, mindenfajta algoritmus nélkül. A megismerésük viszont újabb kihívásként szolgált a szakdolgozatom során.

A *P4Graph* állomány három osztályból áll.

A *P4Vertex* osztály a gráf csúcsainak reprezentációja. A mezői a csúcsok tulajdonságait írja le, melyek a név, az azonosító, valamint a szín.

A *P4Graph* osztály a *QuickGraph* *BidirectionalGraph* nevű generikus osztályából származik. Ezt a *P4Vertex* típusú csúccsal és egy általános éllel teszi meg. A kirajzolásra kerülő fa egy ilyen objektum lesz.

A *P4GraphLayout* a *GraphSharp* *GraphLayout* generikus osztályából származik. Ennek paraméterei a *P4Vertex* csúcs, az általános él, mely *P4Vertex* csúcsokat köt össze és a *P4Graph* gráf. Ez az osztály a fa kirajzolására használt vezérlő miatt került implementálásra.

#### VerificationViewModel.cs

A *ViewModel* réteg fő állománya a *VerificationViewModel*.

##### Mezők

A *VerificationModel* példányán kívül minden privát adattaghoz tartozik egy getter/setter mező. Minden mező értéke kötve van a *View* valamilyen vezérlőjének tulajdonságához.

* Az *ErrorMessage* a felületen megjelenő hibaüzenet szövegét tartalmazza. Ennek szövegét mind a *Model* réteg eseményeken, mind a *ViewModel* réteg parancsokon keresztül állítja.
* Az *ErrorBorder* a hibaüzenethez tartozó szövegdoboz szegélyének vastagságát állítja. Ez alapértelmezetten 0, vagyis nem látszik, de ha az *ErrorMessage* értéke nem egy üres szöveg, tehát hiba történt, akkor az értéke 3 lesz, így jobban kiemelve az üzenetet.
* A *CalculatedEnvironments* a modelltől megkapott kiszámított környezetek, vagyis *IdEnvironment* objektumok listája. Az ebben tárolt állapotokból lesz felépítve a gráf, mellyel az ellenőrzés folyamata szemléltetve van.
* Az *InitEnvironments* lista tartalmazza a program parser részéből kinyert kezdőállapotokat, amelyekből a kiszámított környezetek kiindultak a számítás elején. Ezek egy listadobozban vannak megjelenítve a felületen, és a felhasználónak választania kell egyet a gráf kirajzolása előtt.
* A *SelectedInitEnv* a kiválasztott kezdőállapotot fogja tartalmazni. Ez lesz a kirajzoláskor a fa egyetlen gyökere.
* A *FinalEnvironments* lista a modelltől megkapott két végállapotot tartalmazza. Ezek meg vannak jelenítve a felületen egy listában, hogy a felhasználó láthassa milyen környezetekkel voltak összehasonlítva a kiszámított végállapotok.
* A *VerificationViewModel* tartalmaz még minden mellékfeltétel osztályból egy példányt. A *SelectConds*, *TableConds*, *AssignmentConds*, *SetHeaderConds* és *DropConds* mind a mellékfeltételek megjelenítéséhez, és azoknak módosításához kellenek.
* Az *Input* mező a P4 programot tartalmazza. Az értéke egy szövegdobozhoz van kötve, amely mindig figyeli a szöveg változásait, így folyamatosan frissítve azt.
* A *Summary* az esetleges nagyobb fák kirajzolásakor előforduló átláthatatlanságot küszöböli ki. Az összegzés viszont minden kirajzoláskor megjelenik, függetlenül a fa méreteitől.
* A *GraphToVisualize* a környezetekből felépített gráfot tartalmazza. Ez a *GraphLayout* vezérlőhöz van kötve, így ennek az értéknek a változásakor a fa újrarajzolódik.
* A ConditionString a mellékfeltételek számozásainak szöveggé alakításának értékét tárolja.

##### Parancsok

A *ViewModel* és a *View* közti kommunikáció parancsokkal van megoldva a rétegek közti elkülönülés megtartása érdekében.

### View

### Applikáció

1. Cikk linkje [↑](#endnote-ref-1)
2. A könyvtár dokumentációjának linkje

   <https://hackage.haskell.org/package/parsec-3.1.14.0/docs/Text-ParserCombinators-Parsec.html> [↑](#endnote-ref-2)
3. Egyszerű imperatív nyelv elemzése

   <https://wiki.haskell.org/Parsing_a_simple_imperative_language> [↑](#endnote-ref-3)