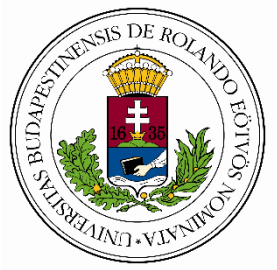


Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok
Tanszék

P4 programok helyességének ellenőrzése bővíthető szabályrendszer alapján

Témavezető:

Tóth Gabriella
Doktorandusz

Szerző:

Nagy Rebeka
Programtervező Informatikus BSc

Budapest, 2020

Tartalomjegyzék

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 programozási 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 íródott, 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 információt dolgozza fel.

1.1 A használt módszer alapja

A hiba észlelés működésének a Tóth Gabriella és Tejfel Máté által szerzett cikk[1] az alapja. Ez egy axiomatikus szemantikához hasonló szabályrendszer, amelynek összetettebb a környezeti struktúrája, és mellékfeltételekkel van kiegészítve.
A szakdolgozatom a fent említett szabályrendszert csak alapul veszi, nem azt implementálja. Az ellenőrzés ehelyett a program működésének szimulációja, mely környezetek átírásán alapul. A program viselkedése a környezetek változtatásával van reprezentálva és végig követve, a szabályok pedig az azokhoz rendelt feltételek mellett hajtódnak végre.
Ezeket a feltételeket tudja a felhasználó módosítani, ezzel bővítve a programon végzett ellenőrzéseket.

1.2 A P4 programozási nyelv[2]

A P4[3] egy olyan programozási nyelv, amelynek feladata hálózati csomagok feldolgozása és továbbküldése. Jelenleg két nagyobb verziója van a $P4_{14}$[4] és a $P4_{16}$[5]. Szakdolgozatom során utóbbival foglalkozom.
Szerkezetileg három nagy részre bonthatjuk. Ezek a *parser*, *deparser* és a kontrollfüggvények által meghatározott adatmódosító.

Parser

A Parser a bejövő bitsorozatból nyeri ki a csomag információt. A csomagformátum, valamint a csomagokban található mezők leírására a fejléceket használjuk. A parser a fejlécek kicsomagolását végzi, amely azt is meghatározza, hogy mely fejlécek lesznek az adatmódosító fázis kezdetekor inicializáltak.

Az adatmódosító fázis

A P4 programok fő függvénye, mely a program magját tartalmazza.

Ebben legelőször az akciók kerülnek leírásra, melyek kódrészleteket tartalmazó függvények. Ezek például megváltoztathatják egy mező/fejléc inicializáltságát, egy mező értékét.

Ezután a táblák leírása következik. Ezekhez tartoznak kulcsok, valamint előzőleg definiált akciók. A program futásakor ezen akciók valamelyike fog lefutni egy külső vezérlőtől kapott tábla vizsgálata alapján.

Ezután a függvény törzse kerül leírása. Ez az *apply* kulcsszóval jelölt rész határozza meg a konkrét programot, amely mentén kerülnek meghívásra a különböző táblák és akciók.

Deparser

Miután a kontrollfüggvény lefutott, a *deparser* feladata a módosított fejlécek szerint a csomag újra csomagolása. A program végén ez kerül továbbküldésre a hálózaton.

P4 résznyelv

A szakdolgozat alkalmával a P4 programok nem egészével, csak annak egy bizonyos részével foglalkozik az ellenőrzés. Ennek leírása a következő.

A programnak leforduló, P4 nyelven íródott állománynak kell lennie.

A fejlécek definiálására a program legelején kerüljön sor.

A *parser* csak *state* részeket tartalmazhat, amelyeken belül csak fejlécre meghívott *extract()*, *transition* és *transition select* struktúrák szerepelhetnek.

Az *ingress* és *egress* részekben az akciókban nem szerepelhet elágazás, csak értékadás, fejlécekre meghívott *setValid*() és *setInvalid*(), valamint *mark_to_drop()* függvény.

A program *deparser* részében az *emit()* függvényeken kívül nem szerepelhet más, tehát elágazás sem.

Ezen feltételek által meghatározott kisebb és egyszerűbb P4 programokon történhet tehát az ellenőrzés.

1.3 Az ellenőrzés módja

Az ellenőrzés alapötlete, hogy a program minden lehetséges úton, adott kezdőállapotokból milyen végállapotokba tud eljutni, és azok a környezetek megfelelőek-e.

Az állapotnak tehát olyan típusnak kell lennie, mely egyszerűen, de mégis effektíven írja meg, hogy egy adott ponton mi történik a programban. Mivel a nyelv hálózati csomagokat dolgoz fel, és annak fejléceit manipulálja, így erre a legalkalmasabb programrész a fejlécek és annak mezői.

A fejlécek rendelkeznek egy validitás tulajdonsággal, ez egy másik kiinduló pontja az ellenőrzésnek. Az egyszerűbb és konzisztens jelölés miatt a fejlécek mezőinek inicializáltságát is validitással jelöljük.

A P4 nyelv tartalmaz továbbá egy *mark_to_drop()* függvényt, mellyel jelezheti, ha egy csomagot nem akar tovább küldeni. Ennek értékét szintén el tudjuk jelölni validitással, úgy, hogy valid értéket vesz fel az állapotban, ha a csomag eldobásra került.

Az alapul vett módszer olyan típusú hibákat próbál észlelni, amelyek a fejlécek validitásából és a mezők inicializáltságából adódhat. Az ellenőrzés ezt a módszert veszi alapul a programok számításakor. Ebből adódóan, a fentebb leírt információk, vagyis a fejlécek, mezők, a *drop* és ezeknek validitása elegendő a környezetek információdús leírásához.

A P4 program mentén haladva a *parser* részből kinyerhetőek a kezdőállapotok, melynek számossága legalább egy, és az elágazási ágak számától függ. Ha egy fejléc kicsomagolásra kerül ebben a szakaszban, akkor az adott állapotban értékei validak lesznek.

A *deparser* rész határozza meg a végállapotokat. Az egyik végállapotban az újracsomagolt fejlécek értékei validak lesznek. A résznyelv megkötései biztosítják, hogy ebből a környezetből csak egy darab legyen, az elágazások hiánya miatt. A másik végállapot a számítás által generált, amely olyan állapotot tartalmaz, amelyben a csomag eldobásra került, így a *drop* értéke lesz valid.

Az állapotok módosítása a program adatmódosítási fázisában leírt utasítások alapján, a szabályokkal és a beállított mellékfeltételek ellenőrzésével együtt történik. Az ellenőrzés az ezekben megtalálható műveleteket a validitások átállításával szimulálja. Egy *setValid()* függvény esetén például a fejléc értéke az állapotban validra módosul, *setInvalid()* esetén pedig invalidra.

A számítás végén olyan környezetekkel fogunk rendelkezni, amelyek egy-egy lefutási ághoz tartoznak. Hogy megtudjuk melyik állapot helyes, a kezdetben kiszámolt végállapotokkal kerülnek összehasonlításra.

Az összehasonlítás során csak azokat a kiszámolt környezeteket ellenőrizzük, amelyek a szimuláció során nem akadtak el, vagyis a megadott mellékfeltételek mindegyike igaz volt rájuk a számítás során. A környezeteket a validitás értékük alapján vetjük össze.

Ha megegyeznek a végállapotok valamelyikével, akkor a program ahhoz tartozó ága helyes, egyébként pedig nem várt viselkedést okozhat.

A szakdolgozat célja tehát egy olyan alkalmazás készítése, amely a megadott P4 résznyelvnek megfelelő program fent leírt állapotvizsgálaton alapuló elemzését végzi el.

2. Felhasználói dokumentáció

2.1 Telepítés

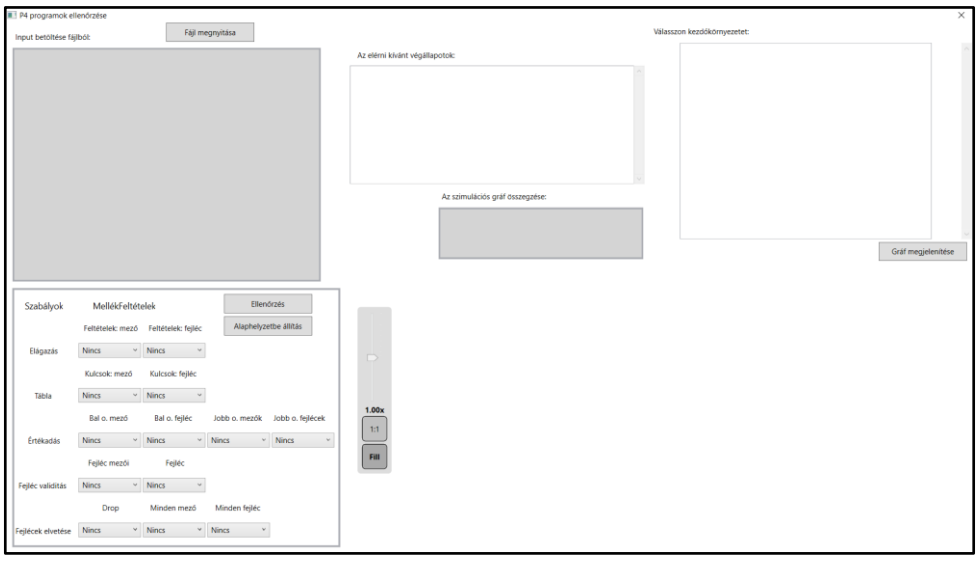
A program nem igényel telepítést, működése teljesen önálló, nem igényel más szoftvert. A főkönyvtárban lévő .\Program almappában megtalálható a futtatható állomány. Erre kettőt kattintva elindul a program.

2.2 Célközönség

A program elsősorban olyan emberek számára ajánlott, akik jártasak a P4 programozási nyelvben, munkájuk vagy szabadidejük alatt ebben a nyelvben implementálnak programokat. A program segítséget nyújthat kisebb P4 programokban előforduló hibák észlelésében, amik könnyen elvéthetőek a felhasználó által. A programhoz alapvető számítógépes felhasználói tudás is szükséges.

2.3 A program

A program egy P4 állomány és a felhasználó által megadott mellékfeltételek beállítása alapján, szimulálva a program működését, megadja azokat a lehetséges futási eredményeket, amelyek helyesek, vagy nem várt viselkedést okozhatnak.



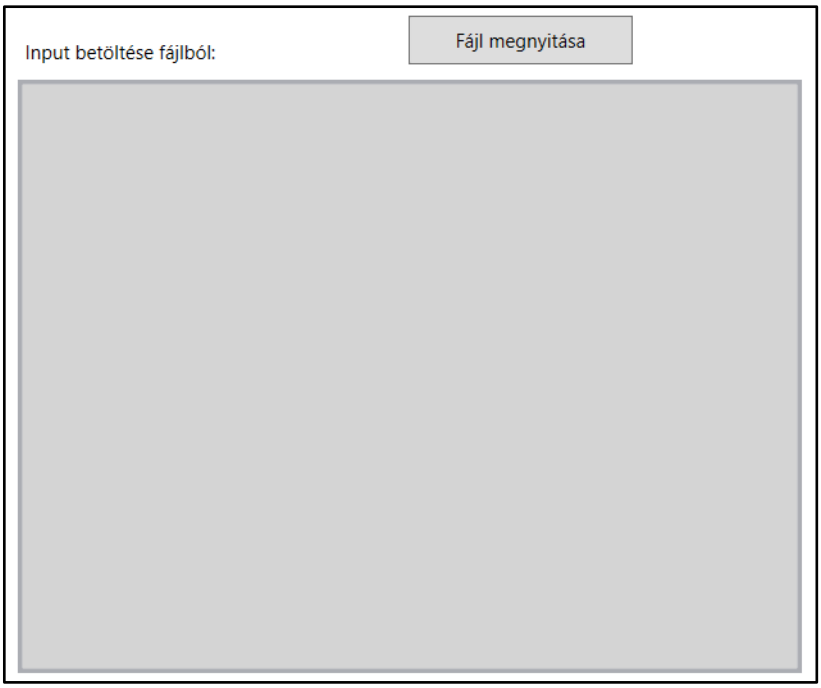
1. ábra - Indítás utáni programfelület

A P4Checking.exe futtatható állományra való duplakattintás után az 1. ábrán látható felület jelenik meg. Ez a programhoz tartozó összes funkciót tartalmazza, nincsenek további ablakok/oldalak/fülek.

A felület részei

A felület öt kisebb részre bontható, melyek a funkciójuk alapján tartoznak össze.

Program beolvasása és szerkesztése

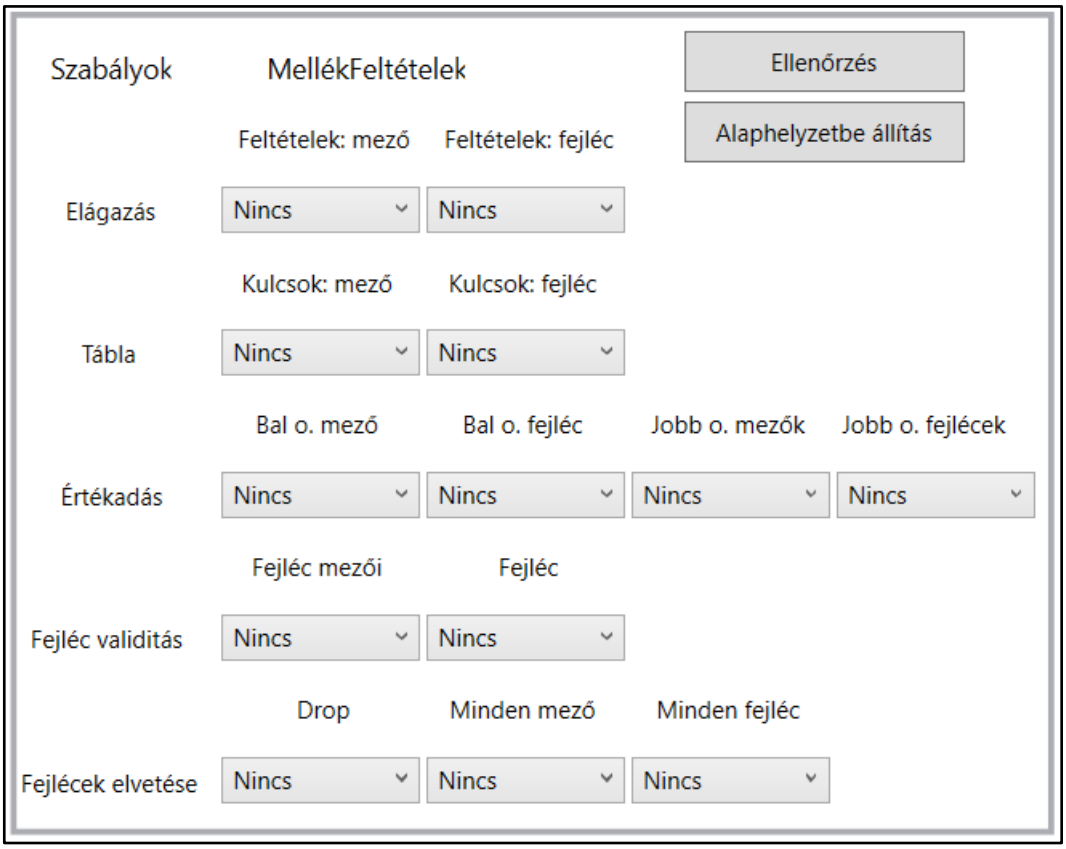


2. ábra - A P4 programok beolvasására és szerkesztésére használt gomb, illetve szövegdoboz a felületen

A bemenő programot a 2. ábrán található szövegdobozban lehet kézzel bevinni. Lehetőség van a program beolvasására fájlból is, mely a „Fájl megnyitása” gombbal történik. A gombra kattintva felugrik egy dialógus ablak, amelyben kiválasztásra kerülhet a program.

A program csak .p4 kiterjesztésű állományokat enged beolvasni. Ha ilyen módon nyitottuk meg a fájlt, akkor a szövegdobozban ezután is lehetőség van a programot szerkeszteni.

Mellékfeltételek és ellenőrzés



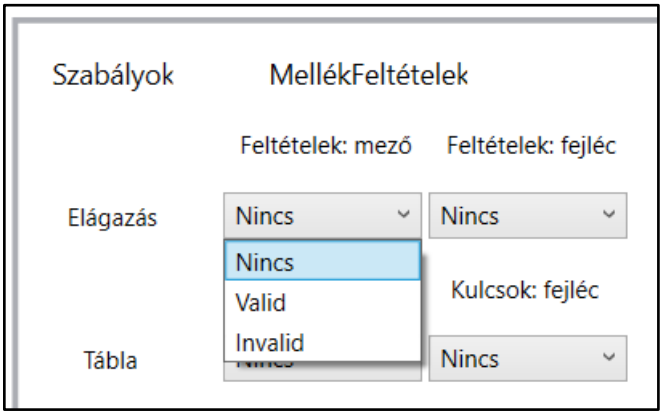
3. ábra - A mellékfeltételek beállításaihoz szükséges legördülő menük, valamint az ellenőrzés és alaphelyzetbe állítás gombok.

A P4 program ellenőrzése előtt lehetőség van a szabályokat oly módon módosítani, hogy hozzájuk tartozó mellékfeltételeket adunk meg, melyek a program helyes működését próbálják ellenőrizni. A szabályok specifikusabb megadására használhatóak, például az értékadás bal oldalára történő *Valid* mellékfeltétel beállítása azt jelenti, hogy az ellenőrzés csak inicializált változót enged majd módosítani. Ezek a 3. ábrán látható módon vannak elhelyezve a felületen.

Minden szabály egy-egy programstruktúrához van kötve, ezek a „Szabályok” oszlopban találhatóak meg. Tőlük balra, sorban helyezkednek el a lehetséges mellékfeltételek.

- Az elágazáshoz megadhatjuk, hogy ellenőrizze le a feltételében szereplő mezőket vagy fejléceket.
- A táblákhoz tartozó mellékfeltételek a kulcsokban lévő mezőkre és fejlécekre vonatkoznak.

- Az értékadás során ellenőrzésre kerülhetnek az egyenlőség bal oldalán elhelyezkedő mező és fejléc, valamint a jobb oldalán lévő mezők és fejlécek.
- A fejléc validitás a *setValid()* és *setInvalid()* függvényekre vonatkozik. Ellenőrzést adhatunk magára a fejlécre, vagy annak mezőire.
- A *mark_to_drop()* függvényre vonatkozik a fejlécek elvetése. A felhasználó az itt szereplő a *drop*, az összes mező, illetve az összes fejlécre állíthat be mellékfeltételt.



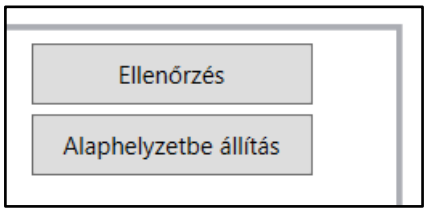
4. ábra - Mellékfeltételekhez tartozó legördülő menük lehetséges értékei

A mellékfeltételek értékeit a hozzájuk tartozó legördülő menüben lehet beállítani, mely az 4. ábrán látható.

A felhasználó választhatja, az alapértelmezett „Nincs" opciót, amely azt eredményezi, hogy a megadott mellékfeltételre nem lesz ellenőrzés.

Ha a „Valid" opciót választja, akkor az adott mellékfeltételben szereplő adat validitását a *Valid* értékkel fogja összehasonlítani.

Az „Invalid" opció az előzővel megegyező beállítás, csak *Invalid* értékkel.

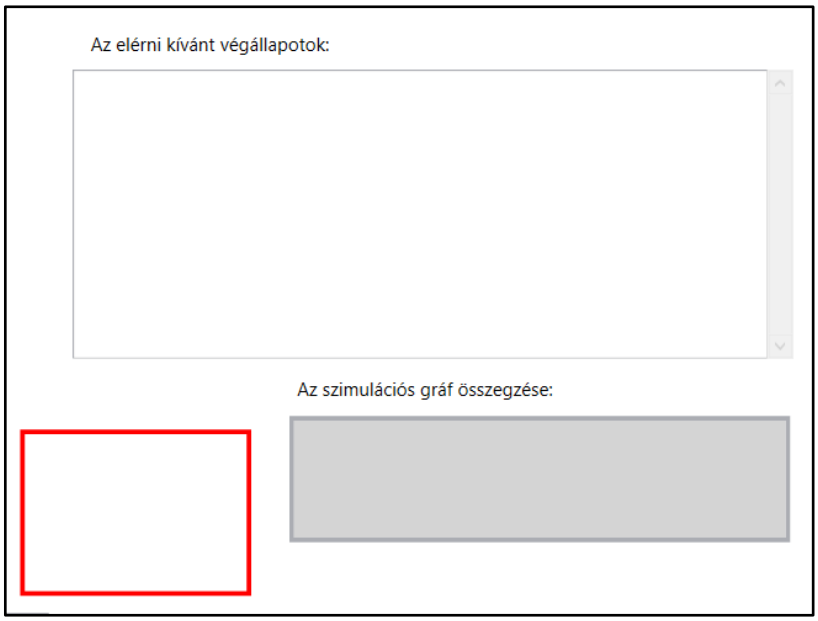


5. ábra - Alaphelyzetbe állítás és Ellenőrzés gombok a mellékfeltételek mellett

A felhasználónak lehetősége van minden értéket visszaállítani „Nincs"-re, a 5. ábrán látható „Alaphelyzetbe állítás" gombbal.

Szintén a 5. ábrán látható a számításokat elindító „Ellenőrzés” gomb. Erre kattintva a megadott program, és mellékfeltétel beállítások alapján megtörténik az ellenőrzés.

Információs panelek



6. ábra - A számítások során fontos információkat megjelenítő felületi részek

A program középső részén három információk közlésére beépített felületi panel található meg. Ezek a 6. ábrán láthatóak.

Az elérni kívánt végállapotokat tartalmazó lista az ellenőrzés után megjeleníti a két környezetet. Ezekkel lesznek a kiszámított végállapotok összehasonlítva.

A szimulációs gráf összegzése doboz a számítási fa leveleit összegzi, és írja ki annak kirajzolása után. Az összegzés módja a következő. A leveleket három kategóriába sorolja.

Az első csoport azok a kiszámított végállapotok, amelyek olyan ág mentén lettek számolva, ahol valamilyen ponton nem voltak érvényesek a megadott mellékfeltételek, így az ellenőrzés ott leállt.

A második csoport azok a környezetek, amelyek olyan ágon haladtak, amik nem álltak le, viszont a végső összehasonlításnál nem egyeztek meg egy végállapottal sem.

A harmadik csoportba azok a környezetek tartoznak, amik lefutott ágat reprezentálnak, és meg is egyeznek valamelyik végállapottal. Az összegzés ezeket számolja össze, és írja ki a felhasználó számára, a könnyebb olvashatóság érdekében.

Ez a rész tartalmaz továbbá egy hibaüzeneteket megjelenítő dobozt, amely nem megfelelő használat esetén figyelmezteti a felhasználót.

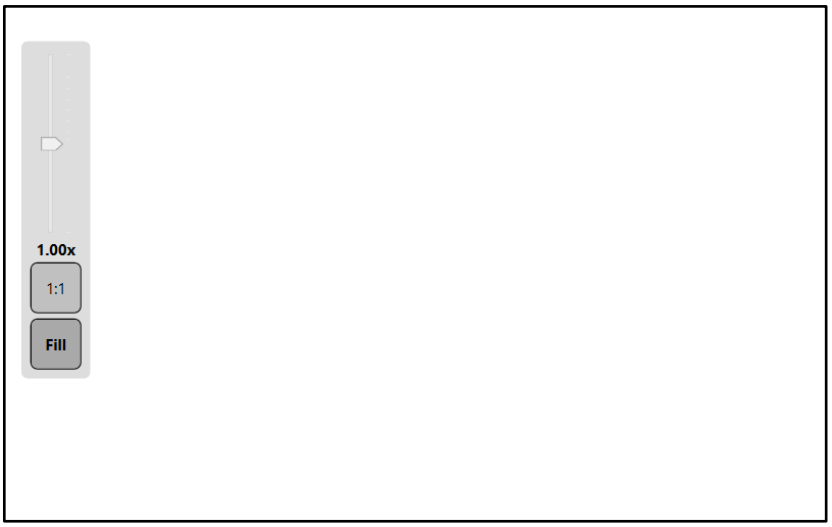
Kezdőkörnyezetek listája és kiválasztása

7. ábra - Kezdőkörnyezetek listája és Gráf megjelenítése gomb

Az ellenőrzés után 7. ábrán látható részben a részben fognak megjelenni a kiszámított kezdőkörnyezetek. Ez egy olyan listanézet, amiben a felhasználó kattintással ki tudja választani az egyik kezdőállapotot. Ennek kiválasztása a szimulációs fa megjelenítéséhez szükséges, mivel az mindig csak egy kezdőállapotból bejárt utakat mutatja meg.

Ha a kezdőkörnyezet kiválasztásra került, akkor a „Gráf megjelenítése” gombra kattintva kirajzolódik az adott mellékfeltételekkel, adott programon végighaladt, adott kezdőkörnyezetből kiindult állapotok gráfja.

Gráf kirajzolásának felülete

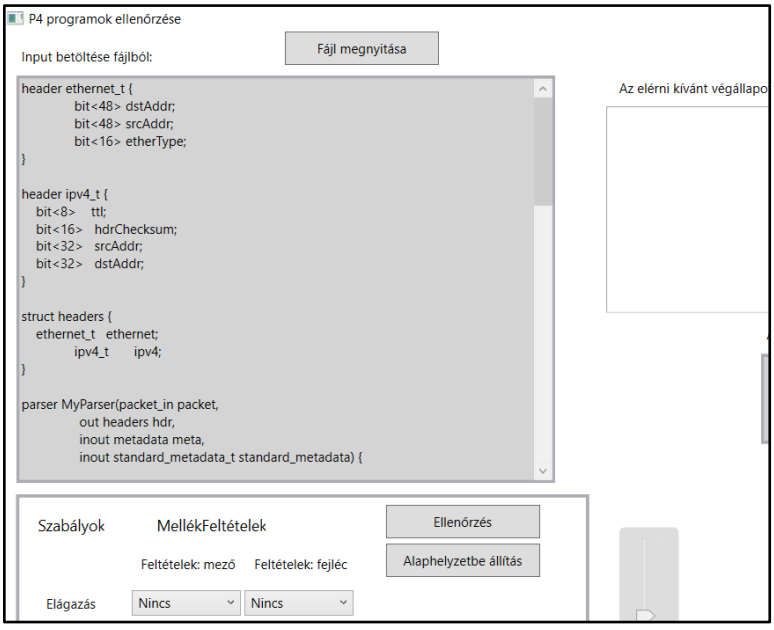


8. ábra - A kirajzolt gráf felülete

A 8. ábrán látható felületen lesz kirajzolva a szimulációs gráf. Ebben lehetőség van a fára nagyítani, vagy kisebbíteni azt. Lehet továbbá tetszőleges irányba elhúzni, egy kattintással egységnyi aránnyal ráközelíteni az „1:1”, vagy alaphelyzetbe állítani a nézetet a „Fill” gombbal.

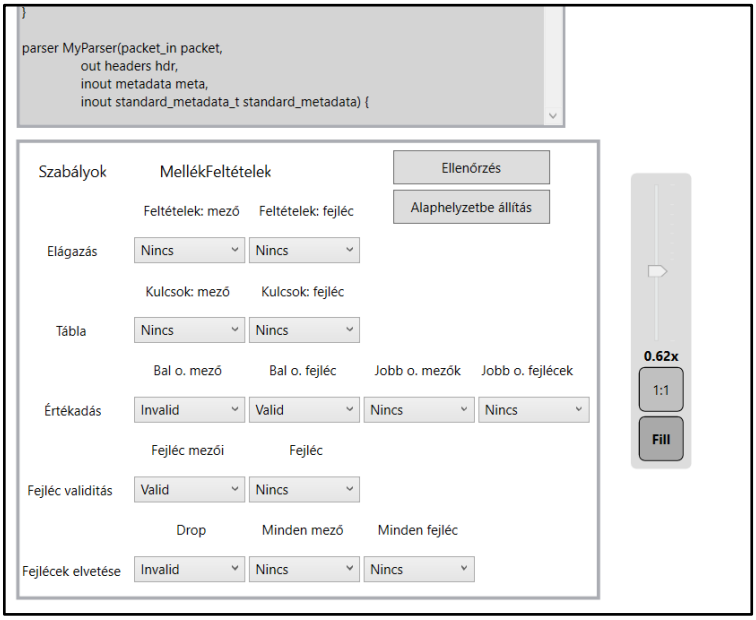
A program használata

A P4 program ellenőrzéséhez első lépésben vagy kézzel kell bevinni a bemenő programot, vagy pedig megnyitni a „Fájl megnyitása” gombbal.



9. ábra - Bemenő program megadása

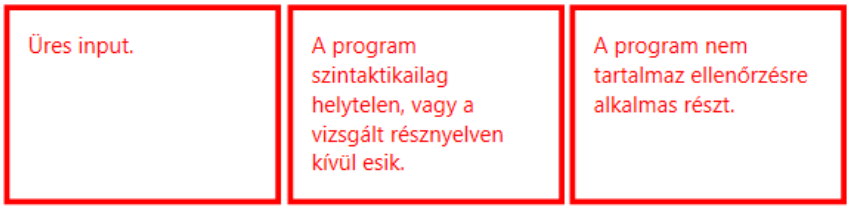
A begépelt vagy beolvasott fájl ekkor a szövegdobozban van, ezt szemlélteti a 9. ábra. Ezután a mellékfeltételek beállítása jön. Ezt a felhasználó tetszőleges módon teheti meg, vagy akár változatlanul is hagyhatja.



10. ábra - Mellékfeltételek beállítása

A 10. ábrán látható egy példa a mellékfeltételek beállításáról. Ha a program szerepel a szövegdobozban, és a mellékfeltételek is be vannak állítva a számunkra megfelelő módon, akkor az ellenőrzés következik. Ennek végrehajtásához rá kell kattintani az „Ellenőrzés” gombra.

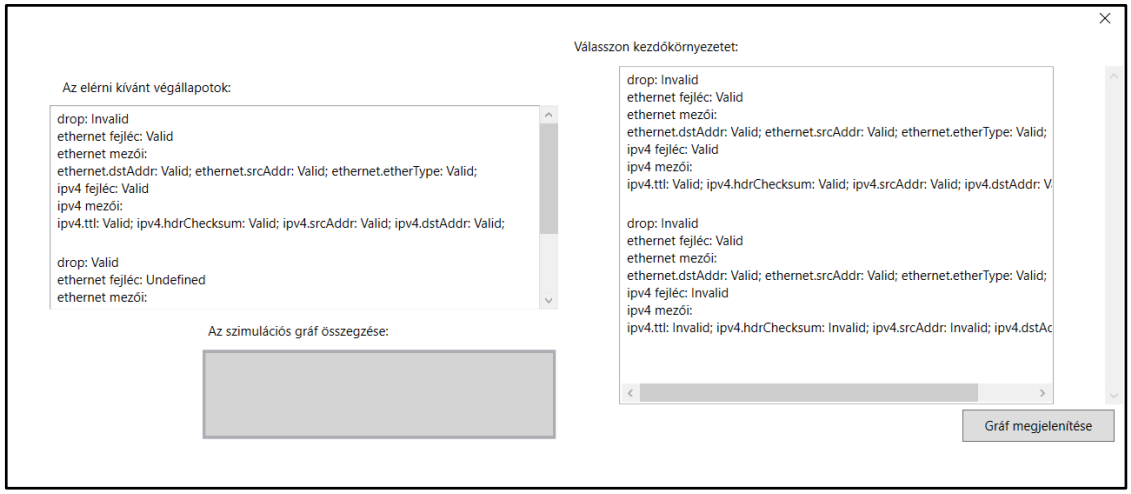
Ha a programunk valamilyen formában nem megfelelő, akkor arra kapunk figyelmeztetést. A 11. ábrán láthatóak azok a hibaüzenetek, amik előfordulhatnak.



11. ábra - Ellenőrzésnél előforduló hibaüzenetek

Az első abban az esetben jelenik meg, ha üres programot próbálunk meg ellenőrizni. A második akkor, ha kisebb szintaktikai hibát vétünk, vagy olyan részeket tartalmaz a program, ami nem tartozik bele a vizsgált résznyelvbe. A program nem egy fordító, így nem minden szintaktikai hibát képes észlelni, ezért előfeltétele az ellenőrzésnek, hogy egy forduló és helyes P4 programot adjunk meg

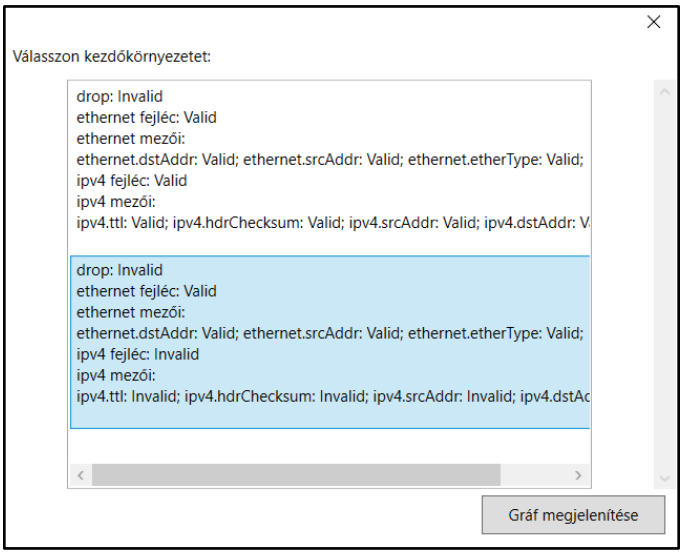
bemenetként. A harmadik hibaüzenet akkor jelenik meg, ha üres a kontrollfüggvény törzse, így nincs értelme az ellenőrzésnek.



12. ábra - Az ellenőrzés után megjelenő adatok

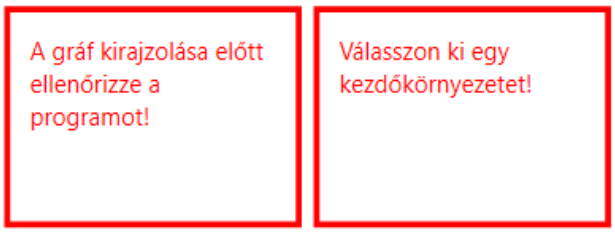
Ha az ellenőrzés során nem történt hiba, akkor a 12. ábrán látható módon jelennek meg az információk a felületen.

Megjelennek a végállapotok, és a kezdőállapotok is. Ezután a felhasználónak ki kell választania a kezdő környezetek listájából azt, ami alapján ki akarja rajzoltatni a gráfot, ahogy ez látható a 13. ábrán is.



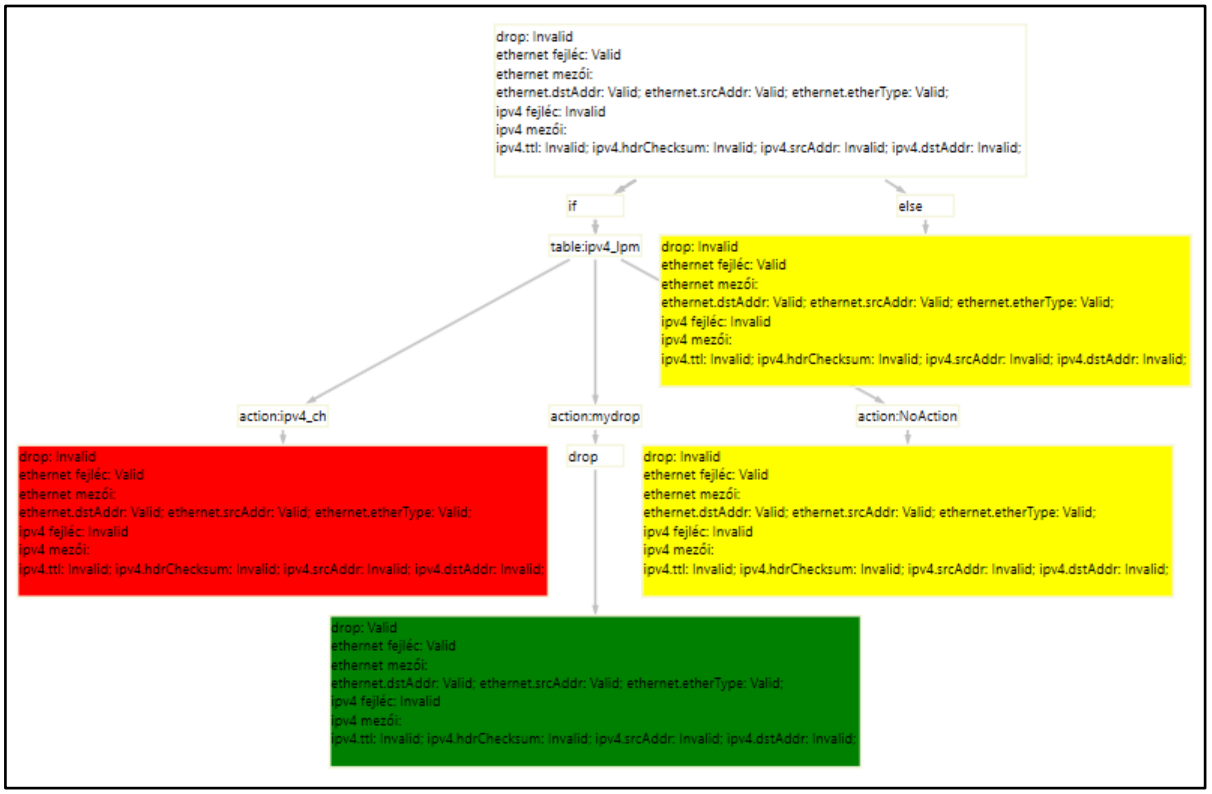
13. ábra - Kezdőkörnyezet kiválasztása a listából

Ha kiválasztásra került a kezdőállapot, akkor a következő lépés a „Gráf megjelenítése” gombra kattintás.



14. ábra - Gráf megjelenítése során előforduló hibaüzenetek

Ha azelőtt kattintanánk a „Gráf megjelenítése” gombra, hogy előtte nem történt ellenőrzés, akkor a 14. ábrán látható első hibaüzenet fogad minket. Ha volt ellenőrzés, de nem került kiválasztásra kezdőállapot, akkor a második hibaüzenet jelenik meg a felületen.



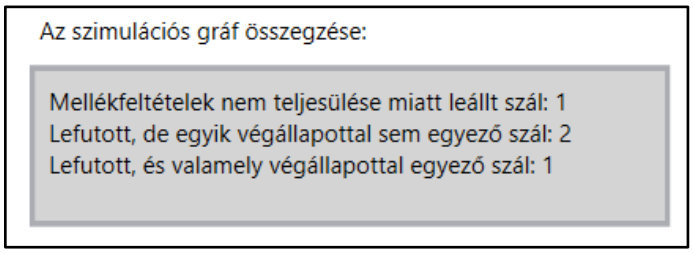
15. ábra - Az ellenőrzéshez tartozó gráf megjelenítése

Ha nem történt hiba a gráf megjelenítése, akkor felületen megjelenik az ellenőrzést leíró gráf, ezt szemlélteti a 15. ábra. A gráf gyökere az a kezdőállapot, amit a felhasználó kiválasztott a listából.

A köztes csúcsok írják le, hogy egy adott állapot fentről lefelé haladva a fában milyen utasítások mentén lett alakítva.

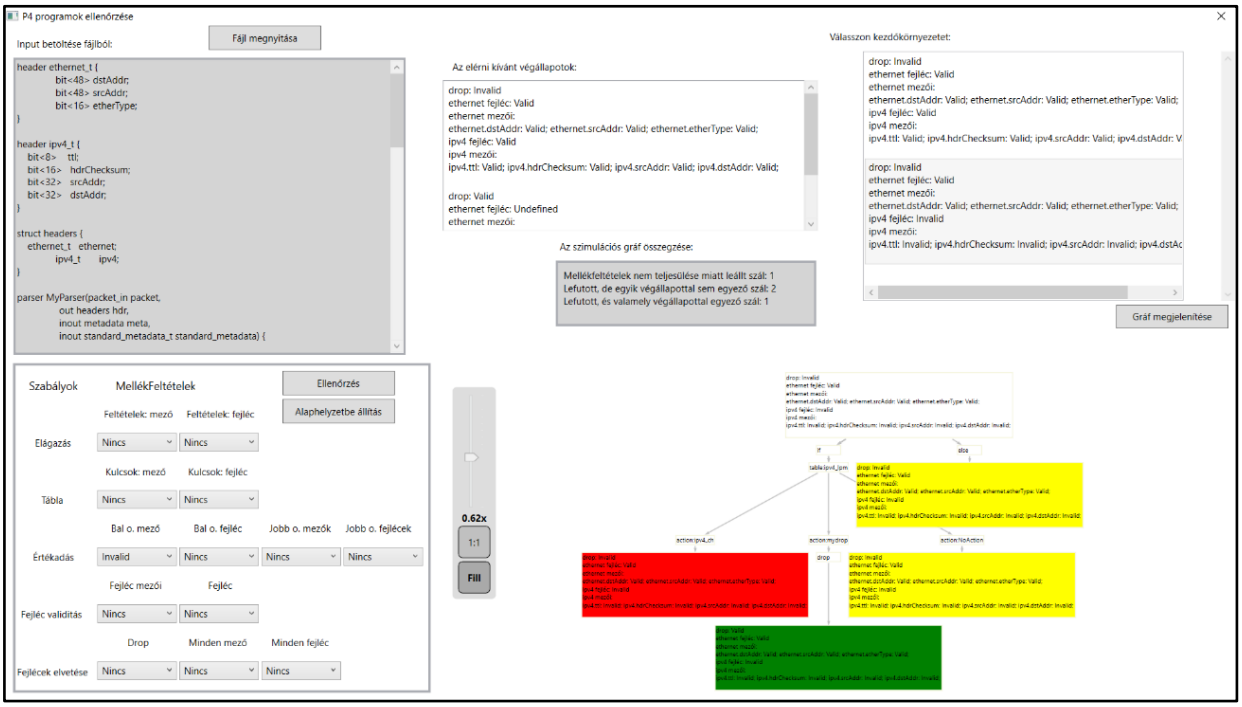
A fa levelei tartalmazzák az összes kiszámolt állapotot.

- A piros levelek mutatják azokat az állapotokat, amelyek megálltak az adott ágon haladva, mert nem teljesült rájuk a felhasználó által megadott mellékfeltételek valamelyike.
- A sárga levelek azokat az állapotokat jelölik, amik nem álltak meg futás közben, de nem illeszkednek egyik végállapotra sem.
- A zöld levelek jelölik azokat az állapotokat, amikre mindig illeszkedtek a mellékfeltételek, és a futás után az összehasonlítások alkalmával megegyezett az egyik végállapottal.



16. ábra - Gráf összegzésének megjelenítése

A felület a 16. ábrán látható módon összegzi, hogy a gráfban milyen levelekből hány darab szerepel, a könnyebb olvashatóság érdekében. Nagyon programoknál ugyanis a levezetési fa túl nagy, és nem teljesen átlátható.

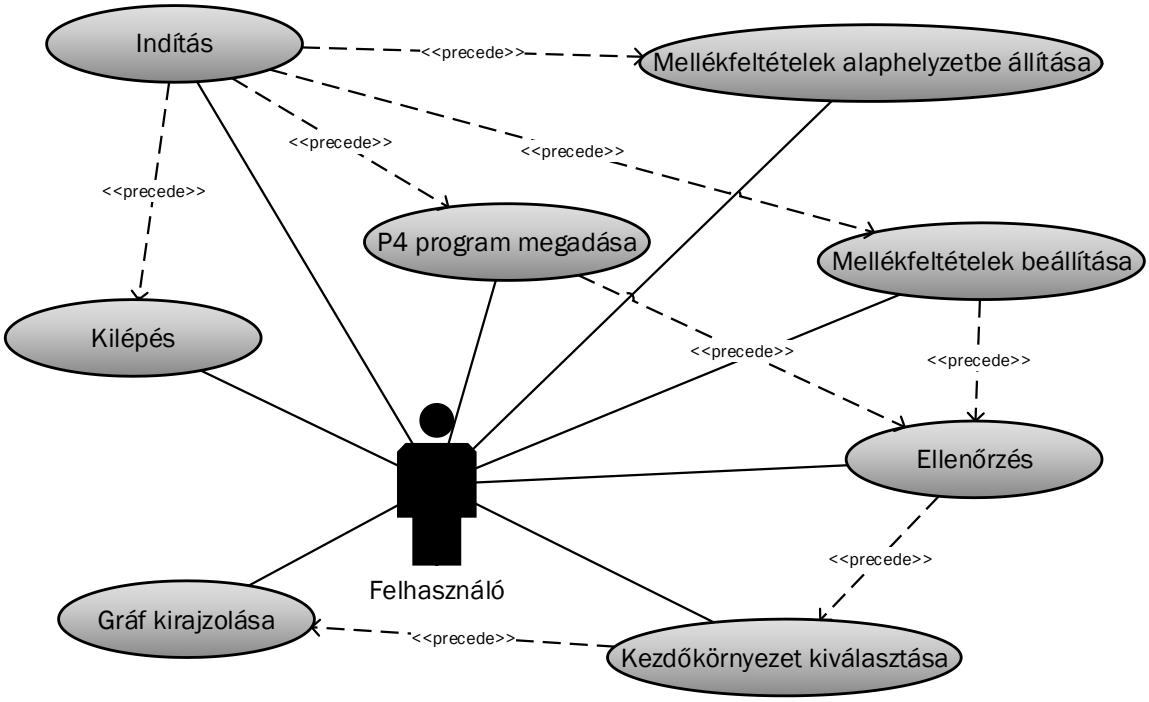


17. ábra - Felület a gráf megjelenítése után

A 17. ábrán látható módon néz ki a felület a folyamatok végeztével.

A folyamatot ezután, és a lépések bármely pontján egyaránt elölről lehet kezdeni, egy másik P4 program, vagy más mellékfeltételek beállításának megadásával. A program ilyenkor törli a megjelenített adatokat, kivéve a gráfot, amely irányt adhat, hogy milyen módon akarjuk módosítani a programot és/vagy mellékfeltételeket.

Felhasználói esetek diagramja



18. ábra - Felhasználói eseteket szemléltető diagram

A felhasználói esetek diagramja a 18. ábrán látható, ami egyszerűen szemlélteti a felhasználó által elvégezhető funkciókat, és azok egymástól függését.

3. Fejlesztői dokumentáció

3.1 Tervezés

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

A környezet

Az állapot tehát olyan módon lesz leírva, hogy tartalmazza a programban előforduló fejléceket és annak mezőit, ezekhez társítson egy validitás értéket, valamint tárolja azt is, hogy előfordult-e a program futása során *mark_to_drop()* függvényhívás. Utóbbi reprezentálására a *drop* változó lesz hozzáadva a környezethez, melynek validitása *Valid,* ha volt ilyen függvényhívás, és *Invalid* ha nem.

A környezet tehát az 19. ábrán látható módon fog kinézni.

drop	drop validitása	[]		
$fejléc_1$	$fejléc_1$ validitása	($mező_{11}$, $mező_{11}$ validitása)	($mező_{12}$, $mező_{12}$ validitása)	...
$fejléc_2$	$fejléc_2$ validitása	($mező_{21}$, $mező_{21}$ validitása)	...	
...				

19. ábra - Állapot ábrázolása

Az így leírt állapotokon könnyen végrehajtható a program szimulációja. Egy programhoz több környezet fog tartozni, mivel a végrehajtás nem lineáris. Minden végrehajtási ághoz tartozni fog egyetlen állapot.

Egy állapot változtatása az akciókban megtalálható utasítások mentén történik. A *setValid()* és *setInvalid()* függvények esetén a megadott fejléc az állapotban lévő validitása fog megváltozni értelemszerűen. Az értékadások esetén a bal oldalon álló mező/fejléc értéke lesz az állapotban *Valid*-ra állítva. Egy *mark_to_drop()* függvény esetén az állapothoz tartozó *drop* értéke *Valid* lesz.

A kezdőállapotokból is több lehet, attól függően, hogy a *parser* részben hányféleképpen csomagolja ki a program a fejléceket. Minden lehetséges mód eredményez egy kezdőkörnyezetet. A kicsomagolt fejlécek egy adott állapotban *Valid* értékkel rendelkeznek, a mezőikkel együtt. Ha egy fejléc nem került kicsomagolásra, akkor a mezőivel egyetemben *Invalid* értéke lesz a környezetben. Az ellenőrzés során a program minden kezdőállapotra hatással van, és bizonyos struktúrák, mint az elágazás, növelik a környezetek számát. Mivel az elágazás esetén az állapotok nem egy lineáris úton, hanem több másik ágon haladnak tovább, így az elágazáshoz elérkezett állapotok az ágak számának megfelelően duplikálva lesznek. A P4 programban előfordulnak az *if-else* elágazások, valamint az ellenőrzés módjából adódóan a táblák is elágazásokként vannak szimulálva, oly módon, hogy minden táblában lévő akció jelent egy ágat.
A végállapotokból kettő lesz. Az egyik a *deparser* rész *emit()* függvényei által van meghatározva, úgy, hogy minden fejléc, amelyre a függvény meg lett hívva, *Valid* értéket fog kapni, mezőivel együtt. A másik egy olyan környezet, amelyben a *drop* értéke *Valid,* és minden más fejléc és azok mező *Undefined* értéket kap.
Az ellenőrzés tehát a kiszámított kezdőállapotokból indul ki, és a program mentén módosítja és bővíti azok számát. A környezetekben mindig azonosak a fejlécek és a mezők, csak azok validitása változik.
A program a szabályokkal és mellékfeltételekkel elvégzett szimulációja után a megkapott környezetek ezután összevethetőek a két végállapottal. Az összehasonlítás a validitás értékek alapján történik, úgy, hogy az *Undefined* értékkel való összehasonlítás minden esetben egyezésnek felel meg. A végső összehasonlítás által ellenőrizhető, hogy a program mely részei okozhatnak nem várt viselkedést.

A program felépítése

Az ellenőrzés implementálására leghatékonyabb nyelvnek a Haskellt[6] tartottam. Ez egy olyan funkcionális programozási nyelv, aminek a mintaillesztésekre épülő

függvényeit tökéletesnek tartottam a szakdolgozat ellenőrzési részéhez. A nyelvről meglévő tudásomat bővítenem kellett a feladathoz, ami újabb kihívást jelentett.

A P4 program ellenőrzéséhez szükség lesz egy olyan részre, amely olyan darabokra bontja szét a programot, amelyből könnyen kinyerhetőek a megfelelő információk, melyek a kezdő- és végállapotok, valamint a program. Ezekből az információkat olyan formára kell hozni, amelyet az ellenőrzés könnyen felhasználhat. Ezután a hiba észlelés rész a megfelelő szabályok segítségével a szimulálja a programot, és módosítja a környezeteket.

A szimuláció végén következhet az a rész, ahol megtörténik az állapotok összehasonlítása. Ezek az állapotok a programból kinyert végállapotok, és a számítással elért környezetek.

Az ellenőrzés rész tehát Haskellben lesz implementálva, viszont a felhasználó felület ebben a nyelvben nagy bonyodalmakat okozna. Napjainkban a Haskell nyelvhez tartozó keretrendszerek, amelyekben felületet lehet fejleszteni, jelentősen bonyolultabbak, mint más módszerek.

Így arra a döntésre jutottam, hogy a felületet valamilyen objektumorientált nyelvben írom meg. A választásom a C#[7] nyelvre esett, mert a tanulmányaim során ebben a nyelvben írt felületek tetszettek a legjobban, és az egyetemen megszerzett tudásomat is kamatoztatni akartam a szakdolgozatom során.

A felület tehát C#, így a keretrendszer a Visual Studio. Az applikációm Windows Presentation Foundation típusú, amely magával vonta a Model-View-ViewModel struktúrát is.

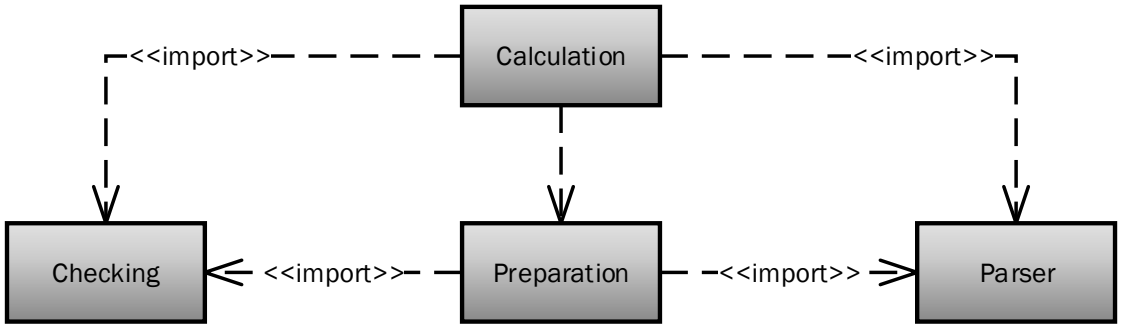
Mivel az ellenőrzés és a felhasználói felület két különböző nyelven és keretrendszerben lesz implementálva, így a köztük lévő gördülékeny kommunikációt is biztosítani kell. Ez úgy valósult meg, hogy a Haskellben megírt ellenőrző rész egy .dll kiterjesztésű állománnyá van alakítva. Így a C# nyelven megírt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

3.2 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.



20. ábra - Haskell komponensek diagramja

Négy állományból áll, melyek az elemző folyamatokat végző *Parser.hs*, az átalakító függvényeket tartalmazó *Preparation.hs*, az ellenőrzést végző *Checking.hs* és az ezeket összefogó és a felhasználói felülettel kommunikáló *Calculation.hs*. Ezek egymást a 20. ábrán látható módon tartalmazzák

A környezetet leíró adattípusok

A környezetek leírásához implementált adattípusok a *Checking.hs* állományban találhatóak meg.

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.

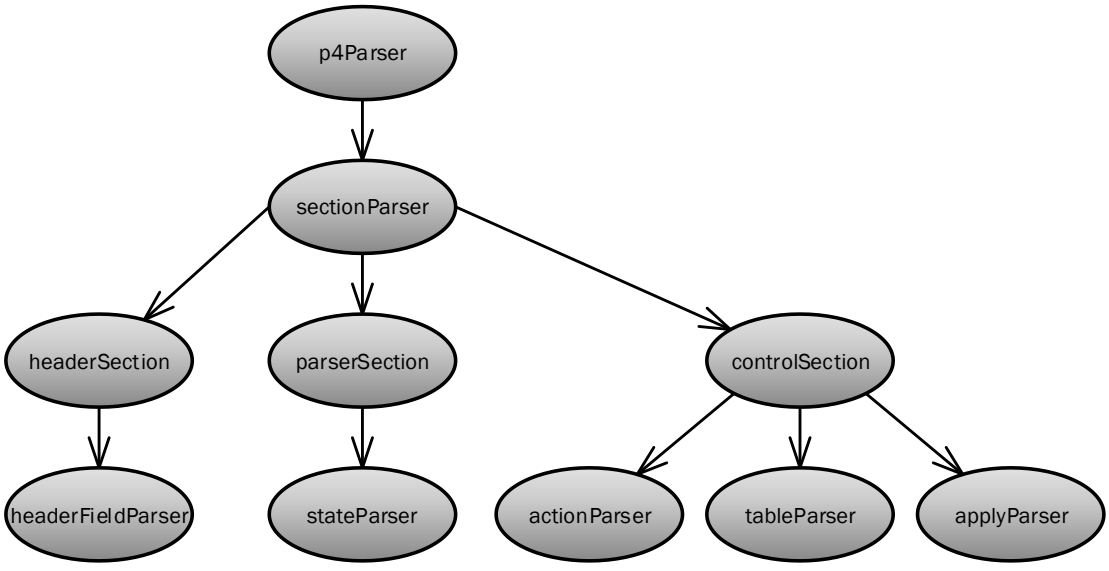
Environment

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.

Parser.hs



21. ábra - A Parser működésének egyszerűsített szemléltetése egy diagrammal

A *Parser* működésének összefoglalása a 21. ábrán látható módon történik.

Az első lépésben a program feldolgozása történik, amelyet a Text.ParserCombinators.Parsec[8] könyvtár, valamint egy egyszerűbb példa[9] segítségével oldottam meg.

A nyelvtani elemzéséhez 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, az *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

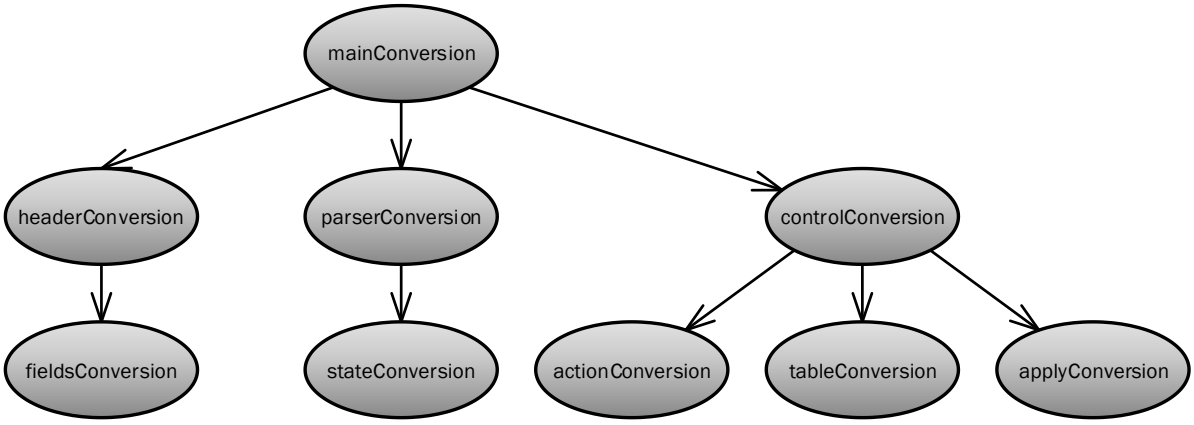
Miután a szintaktikai elemző a szükséges információkat kiszedte a programból, átadja ezt a *Statement* listát az előkészítő lépésnek.

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.

A 22. ábrán látható egy összefoglaló a *Preparation* működéséről.



22. ábra - A Preparation működésének egyszerűsített szemléltetése egy diagrammal

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 *Checking* 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 Checking.hs részben vannak bővebben kifejtve.

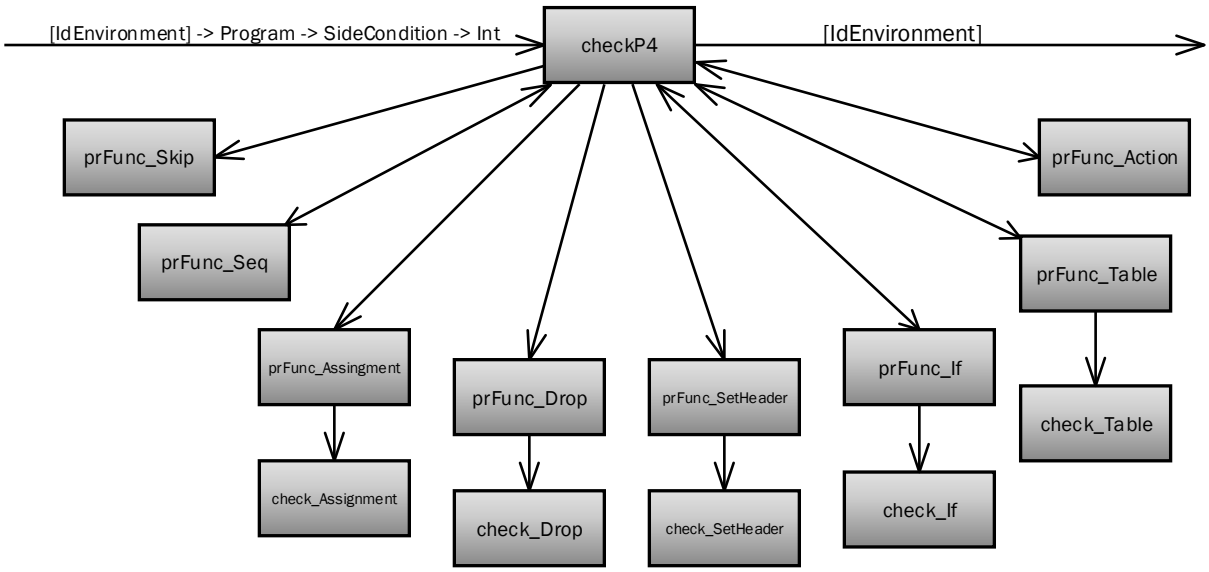
Adattípusokat szöveggé alakító függvények

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.

Checking.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.



23. ábra - A Checking működésének egyszerűsített szemléltetése egy diagrammal

A 23. ábrán látható egy összefoglalás a Checking.hs működéséről.

Bevezetett adattípusok

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, előbbit az azonosítóhoz szükséges számot tízzel, utóbbit hússzal növelve. 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, valamint a százzal növelt azonosítóhoz szükséges számmal 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.

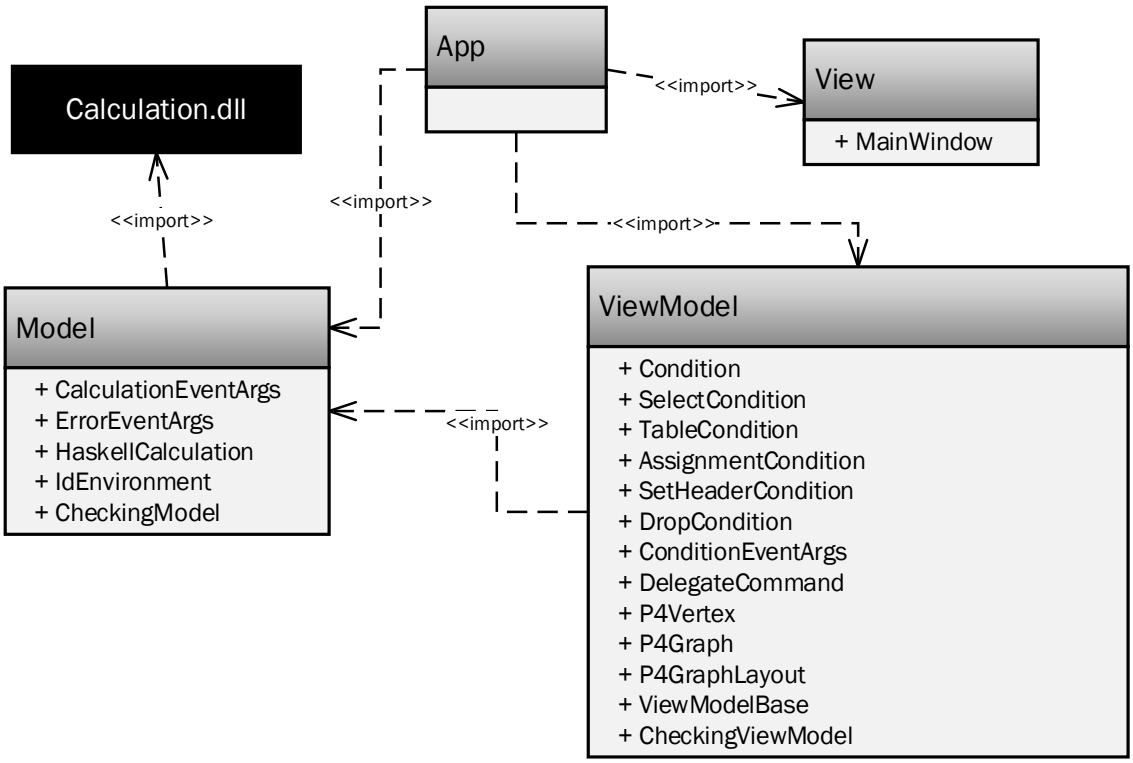
Először a *Parser.hs parseString* függvényét hívja meg a megkapott programmal, amely elvégzi az elemző lépéseket.

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.

3.3 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.
A 24. ábrán látható a komponensek közötti kapcsolat.

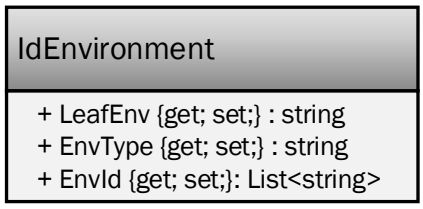


24. ábra – Felhasználói felület komponensek diagramja

Model

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

IdEnvironment.cs



25. ábra - IdEnvironment osztálydiagramja

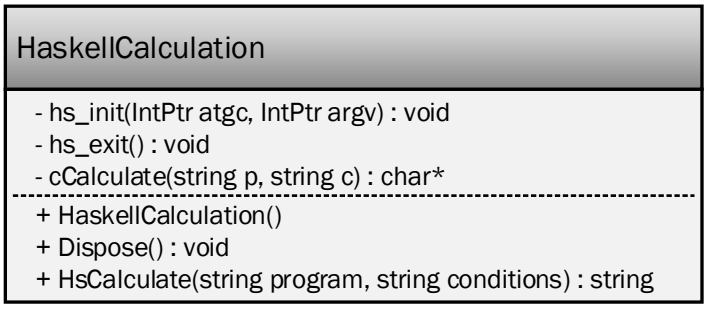
Az *IdEnvironment* osztályt tartalmazó állomány, osztálydiagramja a 25. ábrán látható. Minden az ellenőrző résztől kapott állapot ilyen objektumként van létrehozva. Az osztály az állapotok reprezentálása, melynek három mezője van.

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. Az azonosító lebontva tárolódik, listaként.

HaskellCalculation.cs



26. ábra - HaskellCalculation osztálydiagramja

Ez a fájl a HaskellCalculation osztályt tartalmazza, osztálydiagramja a 26. ábrán látható.

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

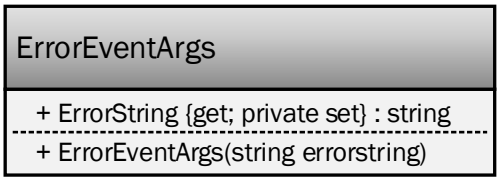
A *Model* és a *ViewModel* közötti kommunikáció eseményeken keresztül zajlik. A modell irányából kétféle információ érkezhet. Vagy a számítás végeztével adja oda az eredményeket, vagy valamilyen hiba esetén hibaüzenetet küld a *ViewModel* felé.

CalculationEventArgs.cs

27. ábra - CalculationEventArgs osztálydiagramja

Ennek az eseménynek három mezője van, melye a 27. ábrán látható osztálydiagramról is leolvashatóak. 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.

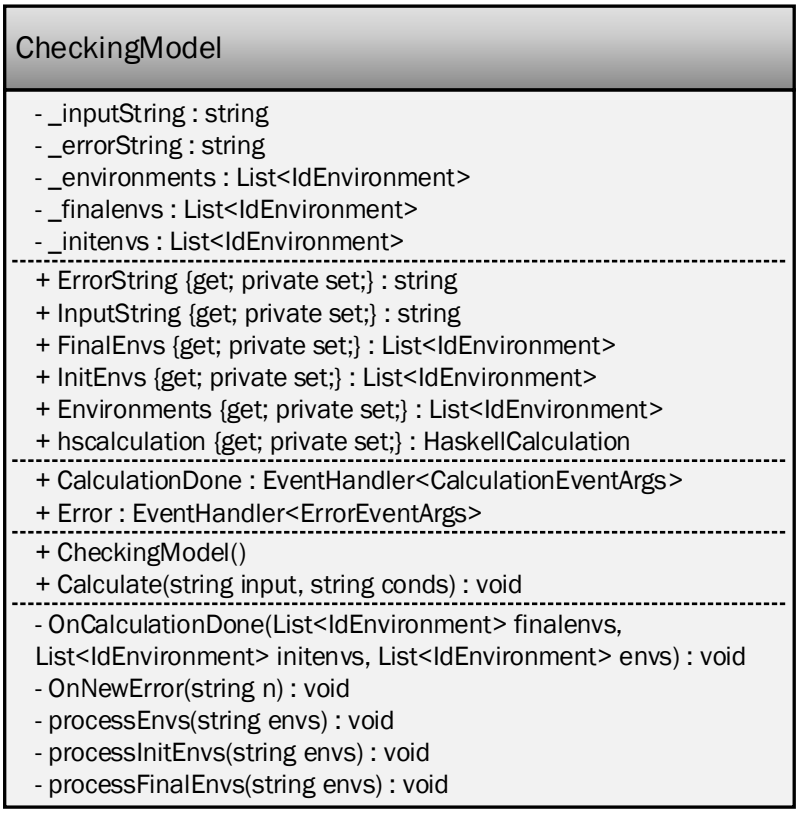
ErrorEventArgs.cs



28. ábra - ErrorEventArgs osztálydiagramja

Az eredményben megkapott hibaüzenetet ez az esemény fogja a ViewModel részére átadni. Ennek tehát csak egy szöveg mezője van, amely magát az üzenetet tartalmazza. Osztálydiagramja a 28. ábrán található meg.

CheckingModel.cs



29. ábra - CheckingModel osztálydiagramja

A modell törzse a *CheckingModel*. Ez az az osztály, amely a *Calculation.dll* könyvtárral folytatja a kommunikációt. Osztálydiagramja a 29. ábrán láthathó.

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.

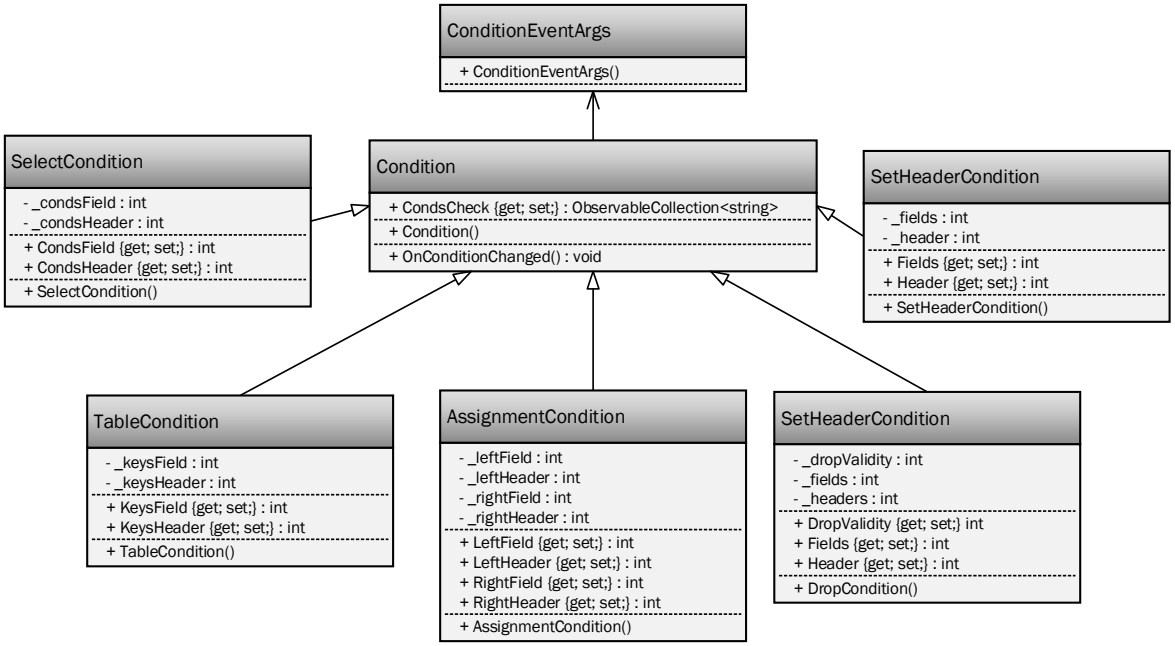
- Az osztály konstruktora paraméter nélküli, és a mezőknek ad megfelelő kezdőértéket.
- A *Calculate* a *ViewModel* által hívott függvény, melynek paramétere a *View* rétegtől megkapott bemenő program és a mellékfeltételek szövegként. Első lépésként az osztály minden mezőjét alaphelyzetbe állítja. Ezután 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



30. ábra - Condition osztálydiagramja, és kapcsolatai

A mellékfeltételek reprezentálásához létrehozott osztály. Ennek segítségével a felületen legördülő menüben szereplő, valamint az ezekben kiválasztott értékek kerülnek tárolásra. Az ősosztály és a leszármaztatott osztályok, valamint az ezekhez tartozó események közötti kapcsolatok leolvashatóak a 30. ábrán látható osztálydiagramról.

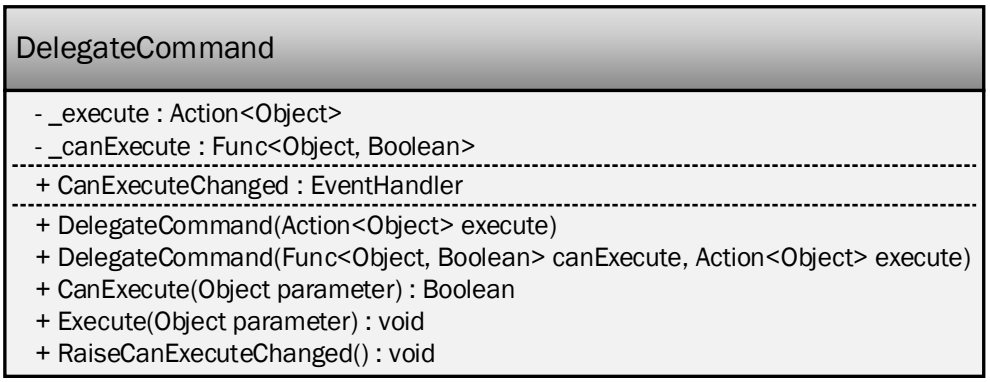
A *Condition* ősosztályban szerepel egy lista, a *CondsCheck*, amely rendre a „Nincs”, „Valid” és „Invalid” szövegeket tartalmazza. Ez a lista az összes legördülő

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 *CheckingViewModel* számára küld jelzést, ha a legördülő menük valamelyike használva volt.

DelegateCommand.cs



31. ábra - DelegateCommand osztálydiagramja

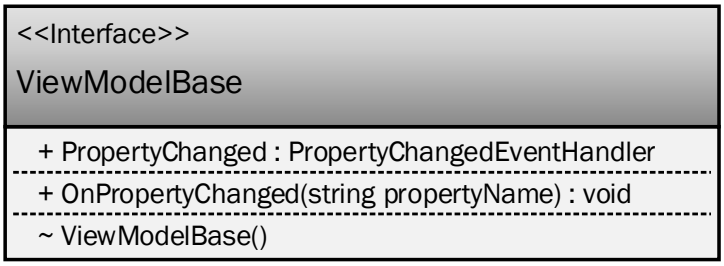
A ViewModel rétegben lévő parancsokat ebből az osztályból származtatjuk, osztálydiagramja megtalálható a 31. ábrán. Mivel az események összekötnék a felületet a modellel ezek parancsokkal vannak helyettesítve.

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.

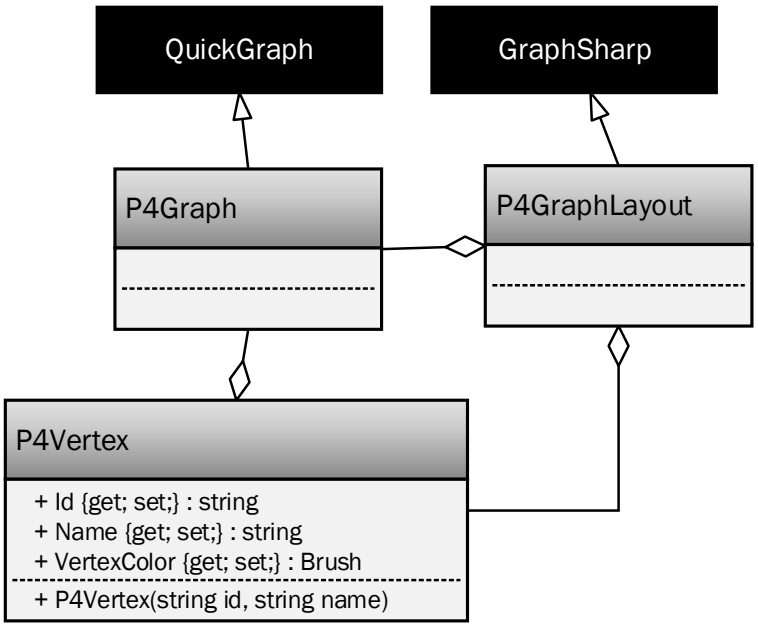
ViewModelBase.cs



32. ábra - ViewModelBase osztálydiagramja

A *ViewModel* változásjelzéssel való kiegészítéséhez egy ősosztály lett létrehozva, mely a *ViewModelBase* osztály. Osztálydiagramja a 32. ábrán látható. Ez az *INotifyPropertyChanged* osztályból van származtatva. Ennek segítségével egyszerűbben jelezhetjük a View felé, ha valamelyik megjelenített érték megváltozott.

P4Graph.cs



33. ábra - P4Graph osztálydiagramja és kapcsolatai

Az eredmények grafikus megjelenítésének módját gráffal oldottam meg. A gráfhoz szükséges osztályok és csomagok a 33. ábráról olvashatóak le.

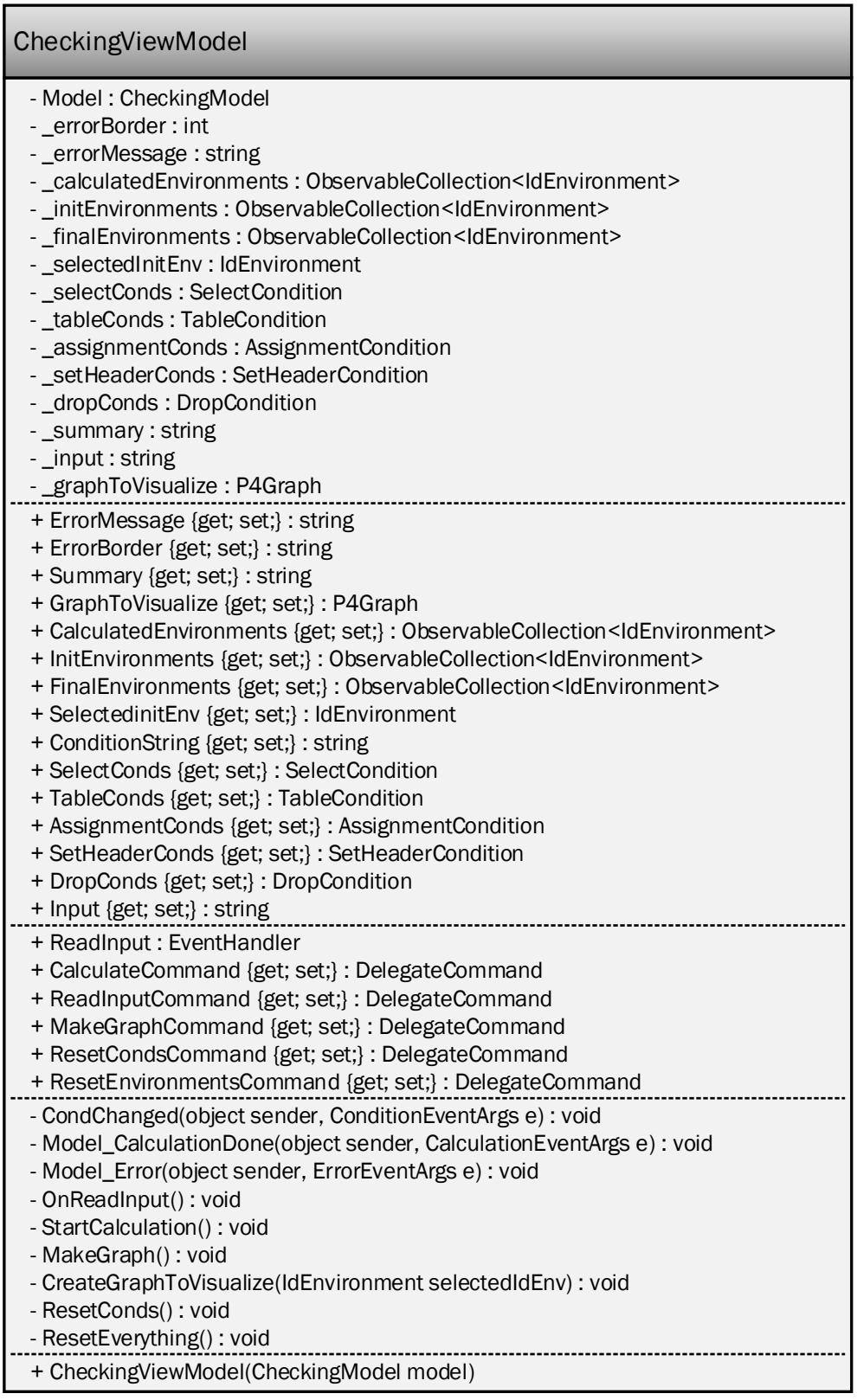
Ehhez egy már létező könyvtárat alkalmaztam, amit *NuGet* csomag formájában adtam hozzá a projekthez. Ennek neve *GraphSharp*[10]. Ez egy olyan könyvtár, amely egy másik, különböző gráfok reprezentálásának és algoritmusainak gyűjteményén alapszik. Ez a *QuickGraph*[11].

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.

CheckingViewModel.cs



34. ábra - CheckingViewModel osztálydiagramja

A *ViewModel* réteg fő állománya a *CheckingViewModel*, melynek osztálydiagramja a 34. ábrán látható.

Mezők

A *CheckingModel* 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 *CheckingViewModel* 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. Ezek mind a *DelegateCommand* osztály objektumai. Bizonyos felületi vezérlők *Command* tulajdonságához vannak kötve.

- A *CalculateCommand* a felületen egy gomb megnyomásához van kötve. Ez az a gomb, amellyel a felhasználó az ellenőrzést elindíthatja. A parancshoz a *StartCalculate* függvény a tevékenysége.
- A *ReadInputCommand* a felületen szintén egy gombra kattintáshoz van kötve. Ez a gomb a P4 program fájlból való beolvasását teszi lehetővé. Mivel ehhez egy újabb felületi ablak megnyitása szükséges, így ez a parancs az *OnReadInput* eseményhez van kötve, amely a *View* számára jelez, ha a gombot megnyomták.
- A *MakeGraphCommand* egy olyan parancs, amely a felületen egy gombra kattintva elindítja a megkapott eredményekből megkapott fa felépítését. A *MakeGraph* függvény a parancs tevékenysége végrehajtás esetén.

- A *ResetEnvironmentsCommand* parancs a felületen megjelenített adatok alapértelmezett helyzetbe való állítását végzik. Ha vagy a program beviteli szövegdobozában, vagy a mellékfeltételek beállításán történik változás, akkor a felületen megjelenő adatok, vagyis a kezdő- és végállapotok, valamint az összegző szövegdoboz ezáltal üressé válik, és nem zavarja össze a felhasználót. A kirajzolt fa viszont megmarad, ezzel mégis követni tudja az előzőleg kiszámolt program lehetséges kimeneteit. A tevékenysége a *ResetEverything* függvény.
- A *ResetCondsCommand* parancs egy olyan gomb megnyomása esetén hajtódik végre, amellyel a különböző beállításokra került mellékfeltételek legördülő menüit egyszerűen alaphelyzetbe lehet állítani. A *ResetConds* függvény a tevékenysége.

Minden parancs létrehozásakor a *param => function()* lambda függvény segítségével rendelem hozzájuk a tevékenységfüggvényeket.

Függvények

A függvényeinek mindegyike a parancsokhoz vagy eseményekhez tartozó függvények.

- A konstruktora egy *CheckingModel* objektumot vár paraméterül. Az osztályhoz tartozó mezők inicializálása, valamint a parancsok és események létrehozása, és tevékenységfüggvényükkel való összekapcsolása a feladata.
- A *Model_CalculationDone* függvény a modelltől érkező eseményhez tartozik, ez jelzi a *ViewModel* számára, hogy az ellenőrzés lefutott, és az eredmények készen állnak a további feldolgozásra. Az esemény paramétereiben megkapott három listát a nekik megfelelő mezőkben helyezi el, melyek a *CalculatedEnvironments*, *FinalEnvironments* és *InitEnvironments*.
- A *Model_Error* szintén a modell egyik eseményétől érkezik, és hibaüzenetet szállít a viewmodell számára.

- Az *OnReadInput* függvény generálja azt az eseményt, amelyet a View saját magának küld a *ViewModel* rétegen keresztül.
- A *CondChanged* egy olyan átvezető függvény, amely a *ViewModel* *Condition* osztályában történő esemény esetén kerül végrehajtásra, és a *ResetEverything* függvényt hívja meg.
- A *StartCalculation* a hiba észlelés gomb tevékenysége, amely először az összes mellékfeltétel legördülő menüjéhez kapcsolt számokat alakítja át, és fűzi össze egy listává és tárolja el a *ConditionString* mezőben. Ezután minden mezőt, mely korábbi számítási eredményeket tartalmazhat alaphelyzetbe állít, és ezután a modell *Calculate* függvényét hívja meg, paraméterként átadva az *Input* és a *ConditionString* mezőket.
- A *MakeGraph* kerül meghívásra, ha a gráf kirajzolására megadott parancs kerül végrehajtásra. Ez először ellenőrzi a *SelectedInitEnv* mezőt, hogy a felhasználó ténylegesen választott-e ki kezdőállapotot, amely a gyökérben kerülne elhelyezésre. Ha ez nem történt meg, akkor egy megfelelő hibaüzenet kerül az *ErrorMessage* mezőbe, egyébként pedig meghívásra kerül a *CreateGraphToVisualize* függvény a *SelectedInitEnv* paraméterrel.
- A *CreateGraphToVisualize* a kirajzolandó fát építi fel a *CalculatedEnvironments* segítségével. A paraméterben megkapott kezdőkörnyezet lesz a fa gyökere. A környezetlista *IdEnvironment* objektumokat tartalmaz, melyeknek egyik tulajdonsága az *EnvId*. Ez egy olyan lista, amely sorban tartalmazza azokat a lépéseket, amiket elvégzett a kalkuláció egy kiszámolt állapoton. Ezáltal a függvény könnyen fel tudja építeni a gráfot.

  A feldolgozás első lépéseként a megkapott *SelectedInitEnv* kerül be a gráfba. Ennek *EnvId* mezője egy egyelemű lista, amely egy számot tartalmaz. A csúcs azonosítója ezzel lesz egyenlő, a neve pedig a *LeafEnv* mezőben eltárolt környezet lesz.

Ezután végighalad a *CalculatedEnvironments* objektumain, úgy, hogy szűrésre kerül a hozzájuk tartozó *EnvId* első eleme. Ha az állapot első eleme ugyanis megegyezik a *SelectedInitEnv* azonosítójával, akkor az a környezet biztosan abból a kezdőállapotból került kiszámításra. Ha ez igaz, akkor végighalad az azonosítót tartalmazó listán. Minden elem belekerül a gráfba csúcsként, úgy, hogy az *Id* maga az azonosító lesz, a név pedig a programstruktúra, de csak akkor, ha az már előzőleg nem szerepelt a fában. Mindig számon van tartva az a csúcs, amely legutoljára került bele, vagy pedig már szerepelt a gráfban és egyezett az új csúccsal. A legújabb létrehozott csúcs, amely bekerül a fába mindig ezzel a számontartott csúccsal van összekötve egy éllel.

Miután a végighaladt létrehozza azt a csúcsot, melynek *Id* és *Name* mezője is az adott *IdEnvironment* objektum *LeafEnv* tulajdonságának értékét kapja meg. Ezek a fa levelei lesznek, az egyetlen csúcsok melynek színezése is van. Ez az *EnvType* tulajdonság értékének megfelelő, *Stuck* - piros, *Match* - zöld és *NoMatch* - sárga. Ez a csúcs végül a legutóbb számontartott csúcshoz lesz éllel összekötve. A feldolgozás pedig folytatódik a következő *IdEnvironment* objektummal.

A csúcsok színezésekor számlálók is alkalmazva vannak, hogy az összegző szövegben pontos érték kerüljön.

Miután kész a fa, értékül adódik a *GraphToVisualize* mezőnek, ami kötve van a felülethez, ezáltal a gráf megjelenik. A *Summary* mező pedig megkapja az összegző szöveget a piros, zöld és sárga színű levelek számával együtt.

- A *ResetConds* függvény egy parancshoz kötött függvény, mely minden mellékfeltételhez kötött legördülő menü értékét alapértelmezettre, vagyis „Nincs”-re állítja.
- A *ResetEverything* függvény bármilyen a bemenő szövegdobozban, vagy a mellékfeltételek legördülő menüiben történő változás esetén alaphelyzetbe

állítja a kiszámított adatokat tartalmazó mezőket, mivel azok már nem lesznek aktuálisak a változtatás miatt.

View

35. ábra - MainWindow osztálydiagramja

A *View* egyetlen *MainWindow* nevű ablakot tartalmaz, ennek osztálydiagramja a 35. ábrán látható. Ezeken különböző *WPF* vezérlők helyezkednek el, XAML nyelven implementálva.

Az bemenő program beolvasásához és szerkesztéséhez egy *TextBox* áll rendelkezése. Közelében megtalálható még egy gomb „Fájl megnyitása" felirattal. Ennek megnyomása egy esemény által fogja megnyitni a fálj megnyitás dialógust.

A mellékfeltételek állításához *ComboBox* vezérlők lettek használva. Ezek mindegyike kötve van az adott *Condition* objektumhoz. Ezekhez tartozik egy gomb, mely alaphelyzetbe állítja őket.

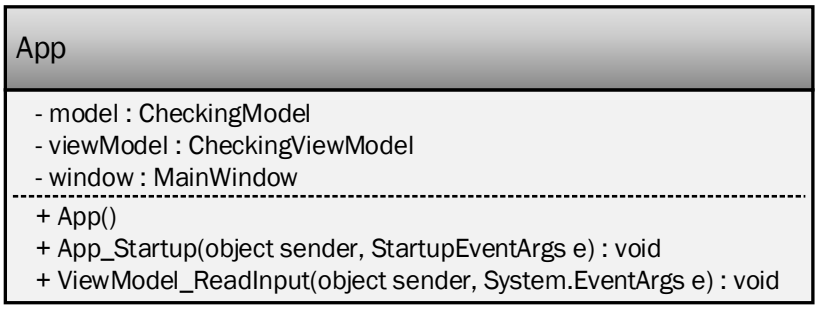
Az ellenőrzés elindításához egy gombra kell kattintani. A kalkuláció paraméterei a gomb megnyitásának pillanatában a felületen megtalálható program és a mellékfeltételek beállításai lesznek.

A vég- és kezdőállapotok megjelenítéséhez *ListView* vezérlő lett használva. Az előbbinél a kiválasztás nem engedélyezett. Mindkettő egy *ScrollView* vezérlőbe van ágyazva a görgethetőség miatt.

Az összegző szöveg, valamint a hibaüzenet is egy-egy *TextBlock* vezérlő. Ezek csak szövegek megjelenítésére vannak használva.

A fa kirajzolásához egy *GraphLayout* vezérlő szolgál, amely egy *ZoomControl* vezérlőbe van ágyazva, ezáltal a fa nagyítható-kisebbíthető és arrébb is lehet húzni. A gráf csak akkor kerül kirajzolásra, ha a felületen elhelyezett „Gráf megjelenítése" gombra kattintanak, miután a kezdőállapotokat tartalmazó *ListView* vezérlőben kiválasztásra került egy környezet.

App



36. ábra - App osztálydiagramja

Az *App* osztály feladata a különböző rétegek példányosítása, a 36. ábrán látható ennek osztálydiagramja. Konstruktorában hozza létre a *View* réteghez tartozó *MainWindow* ablakot, az ehhez kötött *ViewModel* réteg *CheckingViewModel* objektumát, melynek paraméterül adja a már létrehozott *Model* réteg *CheckingModel* objektumát.

Feladata még továbbá, hogy példányosítson egy *OpenFileDialog* párbeszéd ablakot, ha arról eseményt kap a *ViewModel* rétegtől. A felhasználó ezen keresztül adhatja meg a beadni kívánt programot.

3.4 Tesztelés

Hiba észlelés réteg tesztelése

Az hiba észlelés tesztelésére egy külön állomány, a *Test.hs* lett létrehozva, amely az összes modult importálva unit tesztek segítségével ellenőrzi a függvények helyes működését. Az állomány továbbá importálja még a *HSpec*[12] tesztelési keretrendszert, amelynek segítségével a unit tesztek implementálva vannak.

Parser.hs tesztek

Az elemző rész tesztelésére három különböző helyes, valamint egy szintaktikailag nem megfelelő P4 program lett létrehozva. Mind a négy esetet egy unit teszt segítségével ellenőrzünk.

Preparation.hs tesztek

Az átalakító modul tesztjeinek közös jellemzője, hogy a fontosabb függvényeket ellenőrzik, egy bizonyos helyes vagy éppen hibás bemenettel. A tesztek a következőek.

- A *headerConversion* megfelelően alakítja át a fejléceket környezetekké.
- A *parserConversion* helyes bemenet esetén megfelelően módosítja a kezdőkörnyezetek értékeit.
- A *parserConversion* észleli, ha a parser rész nem tartalmaz start állítást, és *EnvError* környezetet adva leállítja az átalakítást.
- A *parserConversion* észleli, ha a *transition* struktúrában a fejléc neve nem egyezik egyik környezetekben lévővel sem, tehát hibás. Ekkor *EnvError* állapotot ad vissza, és leállítja a folyamatot.
- A kontrollfüggvényeket megfelelően alakítja át a *controlConversion* függvény, helyes akciókat, táblákat és programot ad vissza.
- Az *emitConversion* a *deparser* rész alapján helyesen állítja be a végállapotokat.
- Az *emitConversion* észleli, ha az *.emit()* függvény hibás fejléccel van megadva, ekkor *Enverror* állapotot adva leállítja a folyamatot.
- A mellékfeltételeket átalakító függvény a *sideConditionConversion* helyes bemenet esetén megfelelően alakítja át a számsort a Haskell típusra.
- A *sideConditionConversion* helytelen bemenetet észleli, és *SideCondError* mellékfeltételt adva leállítja a folyamatot.
- A *mainConversion* függvény a jól működő részfüggvényei mellett helyes kezdő- és végállapotokat, valamint köztes nyelvi programot ad vissza.

Checking.hs tesztek

Az ellenőrző modul tesztjei elsősorban a programfüggvények helyes működéseit ellenőrzi. A tesztek a következőek.

- A *prFunc_Skip* helyesen a környezeteket változatlanul adja vissza.

- A *prFunc_Action* helyesen módosítja az azonosítót, és az akcióknak megfelelő programfüggvények hajtódnak végre.
- A *prFunc_Drop*, ha nincsenek mellékfeltételei beállítva, akkor helyesen beállítja az összes környezet *drop* értékét *Valid*-ra.
- A *prFunc_Drop,* ha a mellékfeltételeire *Valid*, vagy *Invalid* ellenőrzés van megadva, akkor azoknak megfelelően csak azokban a környezetekben állít át a *drop* értékét.
- A *prFunc_SetHeaderValidity* mellékfeltételek beállítása nélkül minden környezetben beállítja a megadott fejlécet a megadott validitásra.
- A *prFunc_SetHeaderValidity,* ha a mellékfeltételei *Valid*-ra vagy *Invalid*-ra való ellenőrzésre be vannak állítva, akkor csak azokban az állapotokban módosítja a megadott fejlécet a megadott validitásra, ha a mellékfeltételek mindegyike igaz.
- A *prFunc_Assignment* megfelelően működik mellékfeltételek beállítása nélkül, vagyis minden környezetben beállítja a megadott mezőt/fejlécet *Valid*-ra.
- A *prFunc_Assignment,* ha mellékfeltételei be vannak állítva, akkor azoknak megfelelően csak azokban az állapotokban állítja a megadott mezőt/fejlécet, amelyben a feltételek igazak.
- A *prFunc_If,* ha nincsenek mellékfeltételek beállítva, akkor megfelelően működik, vagyis minden környezetet szimulál az *if* illetve az *else* ágon is.
  A *prFunc_If* akkor is megfelelően működik, ha valamennyi mellékfeltétele beállításra került. Ekkor csak azokat a környezeteket duplázza és szimulálja, amelyekre igazak a feltételek.
- A *prFunc_Table* helyesen működik mellékfeltételek megadása nélkül, vagyis elágazásszerűen alkalmazza az akcióit a környezeteken.
- A *prFunc_Table* feltételek mellett is helyesen működik, és csak azokat a környezeteket módosítja, ahol a feltételek igazak.

- A *verifyP4* helyesen működik, és megfelelően adja vissza a módosított állapotokat, ha a programfüggvényekhez nincs beállítva mellékfeltétel.
- A *verifyP4* a programstruktúrákhoz tartozó mellékfeltételek beállítása mellet megfelelően működik és helyes állapotokat ad vissza.

Modulok együttes tesztjei

Az ide tartozó tesztek az előző három modul fő függvényének együttes használatára adott eredményeket ellenőrzi. A három helyes létrehozott P4 programot először elemezzük, majd átalakítjuk, és végül ellenőrizzük. A függvény mindhárom esetben helyes kezdő- és végállapotokat, valamint programot ad vissza.

Ellenőrizve van továbbá az összehasonlító függvény is. A létrehozott három P4 program kiszámított állapotai az összehasonlítás után a megfelelő *EnvType* értéket kapják meg.

Felhasználói felület réteg tesztelése

A felhasználói felület tesztelése manuálisan történt.

- Ha a felhasználó üres programot ad, akkor a megfelelő hibaüzenet erre figyelmezteti.
- A felhasználó csak .p4 kiterjesztésű programokat adhat meg bemenetként.
- Ha a felhasználó a programot módosítja a beolvasás után a szövegdobozban, és ezáltal abban szintaktikai hiba lesz, akkor a program nagy valószínűséggel észreveszi, és azt jelzi is, de az előfeltétele az ellenőrzésnek az, hogy egy helyes és lefordúló P4 programot adjon a felhasználó.
- Ha a beadott program nem rendelkezik ellenőrzésre alkalmas résszel, pl. az *apply* függvénye üres, akkor azt a program jelzi hibaüzenettel.
- A felület bármilyen a programban vagy a mellékfeltételek beállításaiban változás történik, akkor a számított adatok mindig törlésre kerülnek, hogy

mindig egyértelmű legyen milyen programhoz és beállításokhoz tartozik a kiszámított eredmény.

- A fa kirajzolása előtt kötelezően ellenőrizni kell a programot, mivel anélkül nincs mit kirajzolni. Erre hibaüzenet figyelmeztet.
- A fa kirajzolása előtt továbbá kötelező egy kezdőállapotot kiválasztani, hogy a fa gyökere inicializálásra kerüljön.

3.5 További fejlesztési lehetőségek

A programot sokféleképpen lehetne bővíteni és felhasználóbarátabbá tenni.
Ezen ellenőrzés a P4 nyelvnek egy résznyelvével foglalkozik. Ennek kibővítése, és a teljes nyelvre való kiterjesztése egy hasznos fejlesztési lehetőség. Ehhez egy átfogó elemző megírása szükséges, amely a nyelvet teljes egészében felismeri.
Ennek mentén a szabályok bővítése is lényeges fejlesztési lehetőség lehet. Mivel a szabályok a programstruktúrákhoz vannak kötve, a vizsgált résznyelv kiterjesztésével lehetőség van az azokhoz tartozó szabályok, valamint mellékfeltételek bevezetésére. Ezen utóbbiak bár hasznosak, de időigényesek, és szakmai tudásomat aránytalanul kis mértékben növelték volna, így nem kerültek implementálásra.
További lehetőség még egy új felhasználói felület létrehozása az ellenőrző részhez. Az Haskell .dll könyvárrá konvertálható, így akár webalkalmazás, vagy bármilyen más keretrendszerben megírt asztali alkalmazás felületéhez is fel lehet használni.

4. Összefoglalás

A P4 egy, a napjainkban fontos, hálózati csomagok feldolgozására szolgáló eszközök programozására készített programozási nyelv.

A szakdolgozatom célja egy olyan program megvalósítása, amely a P4 programnyelv egy résznyelvével foglalkozik, és ezen programok helyességének vizsgálatát végzi, egy szabályrendszer alapján megírt, állapotvizsgálatokon keresztül végrehajtott, mellékfeltételekkel ellenőrzött programszimuláció szerint.

A szakdolgozatom szerkezetileg két nagy részre bontható, melyekből az egyik a Haskellben megírt elemző-átalakító-ellenőrző részekből álló hiba észlelés, a másik pedig az ehhez tartozó C# nyelven megírt WPF vezérlőkkel MVVM architektúrában implementált felhasználói felület. A két rész közötti kommunikáció a Haskell nyelven megírt modulok dinamikus link könyvtárrá (.dll) való átalakításával van megoldva.

A helyességellenőrzéshez a programból nyerjük ki az állapotleíró objektumhoz szükséges információkat. Ezek a fejlécek validitás tulajdonsága, a fejlécekhez tartozó mezők inicializáltsága, amely szintén leírható a validitás tulajdonsággal, valamint egy *drop* érték, amely a csomag eldobását rögzíti, és értéke szintén jelölhető validitással.

A kezdő környezeteket a program parseréből kapjuk meg. A deparser részből kinyert állapot és a generált, drop-valid környezet alkotja a végállapotokat. A program adatmódosító fázisa tartalmazza a programot, amely mentén a szimuláció lefut a szabályok és mellékfeltételek mellett.

A kiszámolt környezetek a kezdőkörnyezetekből indulnak, és a programban előforduló utasítások módosítják az állapotokban szereplő validitások értékét. A számítás végén ezek a környezetek a programból kinyert végállapotokkal vannak összevetve a validitásuk szerint. Az így kapott eredmények alapján tudjuk megállapítani a helyes, valamint a nem elvárt viselkedést okozó futási ágakat.

Irodalomjegyzék

[1] Gabriella Tóth, Máté Tejfel: A formal method to detect possible P4 specific errors, Position Papers of the 2019 Federated Conference on Computer Science and Information Systems, 19, 2019, [49--56]
https://www.researchgate.net/publication/336071496_A_formal_method_to_detect_possible_P4_specific_errors?fbclid=IwAR1E2LT6AmXX4vBMfORvVz05eU1PLc94LSktdHJOl4BqA_RSoKYmgIEtFEQ 2020.05.30.

[2] A P4 hivatalos oldala
https://p4.org/ 2020.05.30.

[3]: Bosshart, Pat and Daly, Dan and Gibb, Glen and Izzard, Martin and McKeown, Nick and Rexford, Jennifer and Schlesinger, Cole and Talayco, Dan and Vahdat, Amin and Varghese, George and Walker David: P4: Programming protocol-independent packet processors, SIGCOMM Comput.Commun. Rev., 44, 2014, [87—95]
http://dx.doi.org/10.1145/2656877.2656890 2020.05.30.

[4] A $P4_{14}$ hivatalos dokumentációja
https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf 2020.05.30.

[5] A $P4_{16}$ hivatalos dokumentációja
https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf 2020.05.30.

[6] A Haskell dokumentációjának linkje
https://www.haskell.org/documentation/ 2020.05.30.

[7] A C# dokumentációjának linkje
https://docs.microsoft.com/en-us/dotnet/csharp/ 2020.05.30.

[8] A könyvtár dokumentációjának linkje
https://hackage.haskell.org/package/parsec-3.1.14.0/docs/Text-ParserCombinators-Parsec.html 2020.05.30.

[9] Egyszerű imperatív nyelv elemzése
https://wiki.haskell.org/Parsing_a_simple_imperative_language 2020.05.30.

[10] A GraphSharp NuGet csomag hivatalos oldala

https://www.nuget.org/packages/GraphSharp/ 2020.05.30.

[11] A QuickGraph NuGet csomag hivatalos oldala

https://www.nuget.org/packages/QuickGraph/ 2020.05.30.

[12] A HSpec testing framework hivatalos oldala

https://hspec.github.io/ 2020.05.30.