Szakdolgozat

Miskolci Egyetem

Petri háló alapú folyamatmodellezés és alkalmazásai

Készítette:

Hornyák Bence

Programtervező informatikus

Témavezető:

Dr. Kovács László

Miskolc, 2021

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Hornyák Bence (IYWK7C) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Petri hálók, folyamatmodellezés, üzeleti folyamatok, hálók megjelenítése

A szakdolgozat címe: Petri háló alapú folyamatmodellezés és alkalmazásai

A feladat részletezése:

A dolgozat célja egy olyan alkalmazás elkészítése , amely alkalmas Petri hálók folyamainak elemzésére és megjelenítésére. Az alkalmazás inputja a háló leírása egy BPEL folyamatként XML nyelven. Az alkalmazásnak ezt feldolgozni, tagolni és egy Petri háló alapú reprezentációra kell alakítania. A konverziónak parametrizálhatónak kell lennie a Petri háló típusokhoz illeszkedően.

Témavezető: Dr. Kovács László

A feladat kiadásának ideje:

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Hornyák Bence**; Neptun-kód: `IYWK7C` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Petri háló alapú folyamatmodellezés és alkalmazásai* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

A BPEL (*Business Process Execution Language*) nyelv létrejötte elsődlegesen a Web szolgáltatások területéhez kapcsolódik, de a nyelv mint általános munkafolyamat (*workflow*) leíró nyelv, más alkalmazási témakörökhöz is köthető. A BPEL szerepét, fontosságát jól mutatja az a tény is, hogy igen gazdag irodalom található az egyes alkalmazási területekről és speciális szabvány kiegészítésekről.

A BPEL aktualitását jelzi, hogy a megvalósító motorok köre is folyamatosan bővül. Ugyan már lassan 15 év eltelt a szabvány bevezetése óta, a meglévő nagyobb rendszerek (Oracle BPEL Process Manager, IBM WebSphere Process Server, Microsoft BizTalk Server, SAP SAP Exchange Infrastructure) alternatívájaként most is jelennek meg új végrehajtó motor implementációk. A Wikipédia forrása szerint [2] a közelmúltban az alábbi szabad szoftver implementációk születtek (1.1. táblázat).

1.1. táblázat. A BPEL nyelv szabad szoftveres implementációi.

Termék neve	**Fejlesztő**	**Megjelenés éve**	**Licensz**
JBPM	JBoss	2016	Apache
Apache ODE	ASF	2016	Apache
Activiti	Alfresco	2014	Apache

Annak ellenére, hogy napjainkra már több BPEL motor elérhető és használatos, a BPEL szerkesztők és különösen a BPEL validációs rendszerek köre igen szegényes. Ezen tapasztalatokból kiindulva a dolgozat célja egy olyan BPEL validációs rendszer elkészítése, amely a BPEL rendszerek egyik fontos tulajdonságát, a terhelés korlátosságát (*bounded model*) vizsgálja.
A korlátosság azt jelzi, hogy minden csomópontban van egy felső korlát a végrehajtható feladatok számára, intenzitására vonatkozóan. Ha a rendszer nem teljesíti ezt a kritériumot, akkor túlcsordul valamely megmunkáló/tároló helyen. Az elemzés során a korlátosság ténye mellett, a korlát értékei is fontos vizsgálandó jellemzők.

A meglévő tervezői rendszerekben legtöbbször szimulációval történik a főbb paraméterek, a korlátosság vizsgálata. Ezen megközelítésnek rendszerint két problémája van: a vizsgálat teljessége (azaz valóban minden lehetséges esetet áttekintettük-e), illetve a végrehajtási idő (a szimulációk futtatása hosszabb időt is igénybe vehet).

A dolgozatban a BPEL folyamatok Petri háló alapú vizsgálatát végzem el. A Petri háló alapú reprezentáció egy elfogadott és többek által alkalmazott megközelítés. A kidolgozott rendszer inputként egy BPEL modell leírását várja, és kimenetként az elemzés eredményét, illetve a folyamatok nyomkövetését adja vissza. A dolgozatban először bemutatásra kerül a bemenet, tehát a BPEL dokumentum, majd a végtermék a Petri háló és elemzése, majd az, miképp lehet az outputot előállítani.

2. fejezet

A BPEL és folyamatainak bemutatása

A BPEL szabvány alapjai a 2000-es évek közepén jöttek létre az üzleti folyamatok szabványos leírására, elsősorban a Web szolgáltatás (*Web Service*, a továbbiakban röviden WS) alapú környezetre fókuszálva [4]. A nyelv célja definiálni az egyes WS folyamatok vezérlését, koordinálását a kívánt üzleti logika megvalósítására. A BPEL modell magja a folyamatok leírására szolgál, melyekhez együttműködő partnereket szimbolizáló modulok kapcsolódhatnak. A folyamaton belüli lépéseket, végrehajtási algoritmust az aktivitási elemekkel írhatjuk le. A folyamatokhoz változók is rendelhetőek, melyek a kapcsolódó adatkezelést reprezentálják. Az egyes processz modulok üzenetváltással kommunikálhatnak egymással.

A BPEL nyelv a WS környezetből adódóan szorosan kötődik az XML alapú adattároláshoz. A BPEL modell XM állományként áll elő, melyben az egyes séma megkötéseket az XMLSchema szabvány biztosítja. Az XMLSchema nyelv lehetővé teszi az XML dokumentumok strukturális és tartalmi ellenőrzését. A sémanyelv gazdag integritási elem készlettel rendelkezik, és támogatja a típusok származtatását is. Az elemi adatkezelő műveletek, szabályok és kifejezések megadásánál az XPath szabványt kell alkalmazni.

A BPEL nyelv tulajdonságaival és alkalmazási lehetőségeivel számos kutatás foglalkozott már az elmúlt évtizedben. A BPEL nyelv nagy előnye a deklaratív formalizmus, mellyel nem szükséges az egyes modulok kódjába beleégetni az üzleti szabályokat. Az elosztott környezet esetén fontos, hogy az egyes üzleti szabályok illeszthetőek, integrálhatóak és validálhatóak legyenek [17]. A BPEL modell fejlesztése ezen célkitűzések teljesítésére irányult. Több dolgozatban (például [15]) a BPEL nyelv, mint cél modellezési nyelv jelenik meg, s más szabvány folyamat modellezési nyelv, mint UML, BPEL-re történő konvertálását vizsgálják.

A létrehozott BPEL modellnyelv tulajdonságait és alkalmazhatóságát aktívan vizsgálták a 2000-es évek második felében. Ennek keretében a [5] dolgozat az elkészített BPEL modellek dinamikus nyomkövetésére, monitorizására mutat be hatékony algoritmusokat. A modell főbb elágazási és szinkronizációs elemeihez kapcsolódó elemzéseket a [16] dolgozat foglalja össze. A BPEL üzleti folyamatok magasabb szintű, a folyamat egységteszt alapú elemzésére ad javaslatot a [13].

A BPEL nyelv egyik gyakori alkalmazási területe a workflow rendszerek fejlesztése. Több irányban is történtek lépések, hogy a BPEL munkafolyamat modelljét különböző alkalmazási területeken használják. Ezen kísérletek között megemlíthetők a grid és tudományos workflow folyamatok [18] és a mobil alkalmazások fejlesztése [8].

A témakörhöz kapcsolódóan megemlíthető, hogy sokan a BPEL-t tekintették az univerzális workflow leíró nyelvnek, s javaslatok is születtek a BPEL központú workflow szemléletre [19]. A hazai vonatkozású fejlesztések körében kiemelhető a BPEL rendszerek formális validációjára irányuló vizsgálatok köre [11]. A validációs elemzések mellett a modell komplexitás meghatározására is találunk példákat a nemzetközi irodalomban [7].

BPEL nyelv kiterjesztésére is több dolgoztat született, mint például az elosztott rendszerekre történő kiterjesztés [6]. A kiterjesztések körét a [10] dolgozat foglalja össze. Ennek az egyes kritériumait és karakterisztikáját a 2.1. táblázat foglalja össze.

2.1. táblázat. A BPEL kiterjesztéseinek összefoglaló táblázata.

Terület	**Paraméterek**
Kritériumok	Lehetőség a kiszervezésre
	Rugalmasság
	Funkcionalitás
	Fenntarthatóság
	Teljesítmény
	Újrafelhasználhatóság
	Robusztusság
	Használhatóság
Felhasználás	Vezérlési folyam
	Adatintegráció
	Kifejezések és hozzárendelő utasítások
	Nagy mennyiségű adat kezelése
	Szolgáltatás társítása
	Szolgáltatás meghívása
	Változó hozzáférés
Munkafolyam dimenziója	IT infrastruktúra
	Processz logika
	Szervezés
Helye a BPM életciklusban	Modellezés
	IT finomítás
	Statikus analízis, ellenőrzés
	Deployment
	Munkavégzés
	Megfigyelés

3. fejezet

Petri hálók és alkalmazásaik

3.1. A Petri hálók matematikai modellje

A Petri háló egy matematikai leírómodell elosztott rendszerek bemutatására. A modellt Carl Adam Petri készítette. A modell nagyon hasonlít a programozók körében elterjedt folyamat ábrára. A háló irányított élekből, helyekből és átmenetekből (*mint elemek*) áll. Az élek csak két különböző típusú elem között állhatnak. A helyeken jelölő objektumok, úgynevezett tokenek állhatnak. A tokenek csak diszkrét számban fordulhatnak elő egy helyen, és a token átvitele atomi folyamat, azaz nem félbeszakítható. A tokenek elláthatóak attribútummal is, ilyen esetben a tokeneket "kiszínezzük" és színezett petri hálóról beszélünk.

A Petri háló alapvető matematikai modellje egy páros, irányított és súlyozott multigráf $PN(P, T, A, W, S)$, ahol

- $P = \{p_1, p_2, \ldots, p_N\}$: a helyek véges halmaza,
- $T = \{t_1, t_2, \ldots, t_M\}$: egy véges tranzició halmaz,
- $P \cap T = \emptyset$,
- $A \subseteq P \times T \cup T \times P$: az élek halmaza,
- $W : F \Rightarrow N^+$: az élsúlyok halmaza,
- $S : P \Rightarrow N^+$: a kezdőállapot.

3.2. Színezett Petri hálók

Az elemi színezett háló felírható, mint egy

$$CPN(P, T, A, \Sigma, V, C, G, E, S)$$

struktúra, ahol

- $P = \{p_1, p_2, \ldots, p_N\}$: a helyek véges halmaza,
- $T = \{t_1, t_2, \ldots, t_M\}$: egy véges tranzició halmaz,
- $A \subseteq P \times T \cup T \times P$: az élek halmaza,

- Σ: a színek halmazainak halmaza,
- V: a változók halmaza, ahol $\forall v \in V$: változóhoz egy $Type[v] \in \Sigma$ típus rendelhető,
- $C : P \to \Sigma$: a helyekhez színeket rendelő függvény,
- $G : T \to EXPR_V$: az egyes tranziciókhoz kapcsolódó validációs, ellenőrzési kifejezés (logikai értékű),
- $E : A \to EXPR_V$: az egyes élekhez kapcsolódó kifejezés, amely a kapcsolódó hely színhalmazához tartozó értéket vehet fel,
- $S : P \Rightarrow N^+$: a kezdőállapot.

Adott $CPN(P, T, A, \Sigma, V, C, G, E, S)$ színezett hálóhoz az alábbi kezelő funkciók köthetőek:

- $M(p)$: a jelölő (*marker*) függvény, melynek értéke a p helyhez kapcsolódó tokenek halmaza (színezett Petri háló esetén az $M(p)$ elemek színeinek illeszkedni kell a $C(p)$ színhalmazhoz),
- $M_0(p)$: helyek induló tokenkészlete,
- $Var(t)$: a tranziciók viselkedését leíró változók halmaza,
- $b(v)$: az adott v változó értékét megadó kifejezés, ahol $b(v) \in Type[v]$.

Egy t tranzició esetén a $Var(t)$ kifejezés a tranzicióhoz rendelt változók együttese, ahol a változók a $G(t)$ vagy E(a: t-hez kötődő él) kifejezésekben szerepelnek. Az egyes esetekhez tartozó halmazok tehát az alábbiak.

$$Var(t) = \begin{cases} \{n, d\}, & \text{ha } t = SendPacket, \\ \{n, d, success\}, & \text{ha } t = TransmitPacket, \\ \{n, d, k, data\}, & \text{ha } t = ReceivePacket, \\ \{n, success\}, & \text{ha } t = TrancmitAck, \\ \{n, k\}, & \text{ha } t = ReceiveAck. \end{cases}$$

A hálóban egy tranzició akkor engedélyezett (*ready*), ha a bemenő helyeknél a kívánt tokenszám megtalálható. Jelölt hálók esetében:

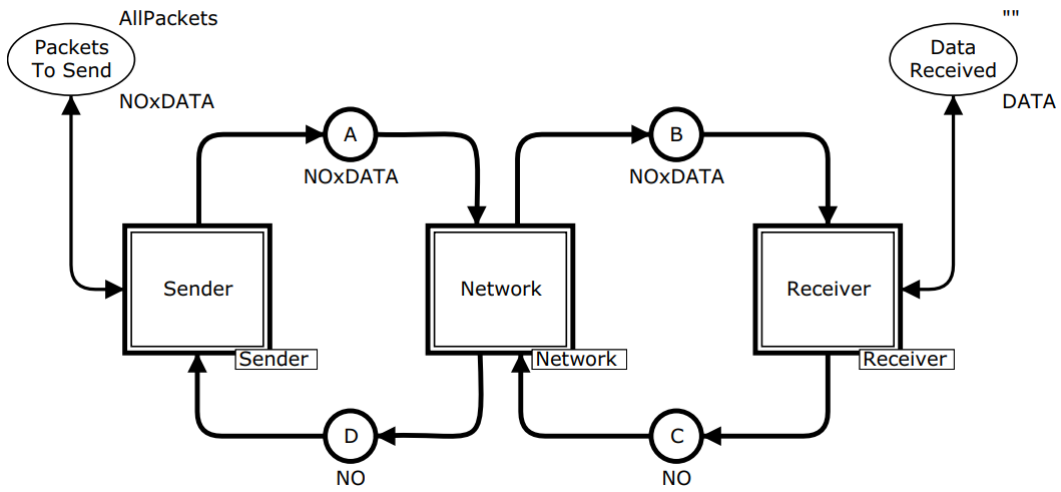
$$M'(p) = M(p) - I(p, t) + O(p, t) : \forall p \in P,$$

ahol

- $I : F \Rightarrow N^+$: bejövő áram, intenzitás,
- $O : F \Rightarrow N^+$: kimenő áram, intenzitás.

A hierarchikus CPN rendszerben az átláthatóság növelése érdekében összefogó modulokat is lehet alkalmazni. Egy modul más elemi egységek együttese, tárolója (3.1. ábra).

A moduloknál fontos szerepet kapnak az átadó helyek, melyeken keresztül a tokenek bejöhetnek a modulba, illetve kiléphetnek a modulból. Az ilyen port jellegű helyek lehetnek bemeneti portok (IN), illetve kimeneti portok (OUT).



3.1. ábra. Rendszerséma 3 modullal

A CPN rendszerek egyik hasznos tulajdonsága, hogy lehetőséget adnak a felépített modell formális ellenőrzésére, validálására és értékelésére. A formális ellenőrzés egyik leggyakoribb eszköze az állapottér (*state space*) modell, ahol az állapottér egy olyan irányított gráf, melyben a csomópontok a háló egy lehetséges $M(CPN)$ jelölési állapota. Azaz a háló struktúrája rögzített, de az egyes elemeknél a tokenek és változók halmaza, azok állapota változhat. A véges állapottér modellt rendszerint szimulációkkal állítják elő.

Az állapottér modellből kiindulva további elemzésekre ad lehetőséget a komponens gráf modell (*Strongly Connected Component Graph*, röviden SCC) formalizmus. Az SCC gráfból a rendszer általános viselkedési szabályaira lehet következtetni. Az SCC gráf olyan gráf, melynek csomópontjai az állapottér azon diszjunkt részhalmazai, ahol egy részhalmaz bármely két elemére igaz, hogy az egyik elem elérhető a másikból.

Az elemzések során az alábbi főbb tulajdonságok elemzésére szokás kitérni:

- Reachability Properties,
- Boundedness Properties,
- Home Properties,
- Liveness Properties,
- Fairness Properties.

4. fejezet

Rendszerterv

4.1. Alapvető feladat elemzése

Egy feladat megvalósítása annak átvizsgálásával és megértésével kezdődik. A feladatkiírás alapján a megvalósítandó programnak tudnia kell

- BPEL file-t olvasni és feldolgozni,
- a feldolgozás eredményeként kapott folyamatot Petri hálóra átalakítani,
- a hálón elemzést végrehajtani,
- a hálón végzett elemzés eredményét, a hálóval együtt megjeleníteni,
- a hálóban zajló folyamatokat lépésenként megjeleníteni.

Az így kapott lista azonban elég vázlatos. Ezeket tovább szükséges bontani.

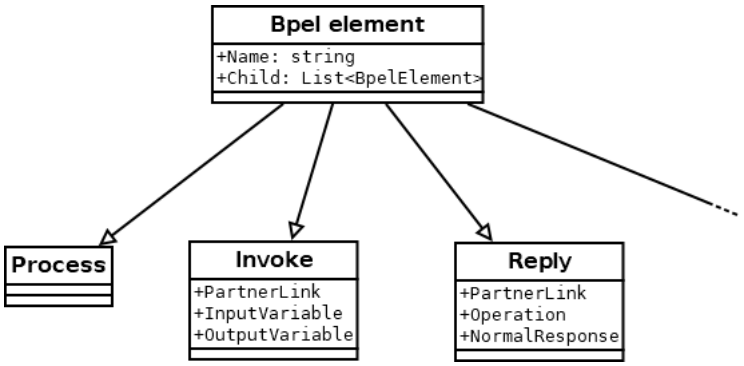
4.2. Feldolgozás

4.2.1. BPEL

Feldolgozás során az olvasott file-t le kell bontani elemeire. Logikus tehát, ha egy file olvasása közben, azzal párhuzamosan a program felveszi egy listára a forrás file-ban szereplő modulokat. Viszont ez idáig csak egy nagyjából futási sorrendű modulhalmaz. A listán szereplő moduloknak tehát még meg kell határozni az attribútumait, illetve, hogy van-e benne visszacsatolás. Ha ezt sikerült meghatározni, csak akkor adható át konverzióra, egyébként a konverzió során nem lenne vezérlési szál, csak különálló hálószegmensek.
Előre láthatólag szükséges egy:

- BPEL element ősosztály, a BPEL elemek definiálásához. Ez fogja tartalmazni az összes BPEL elemre jellemző tulajdonságokat, és leírja a rajtuk végezhető műveleteket (ha van ilyen).
- Külön osztály az egyes elemekre, ami az adott elemet specifikálja.
- Valamilyen adatmodell, ami a vezérlési szálat írja le. Ez akkor szükséges, ha a hálóban visszacsatolás lenne.

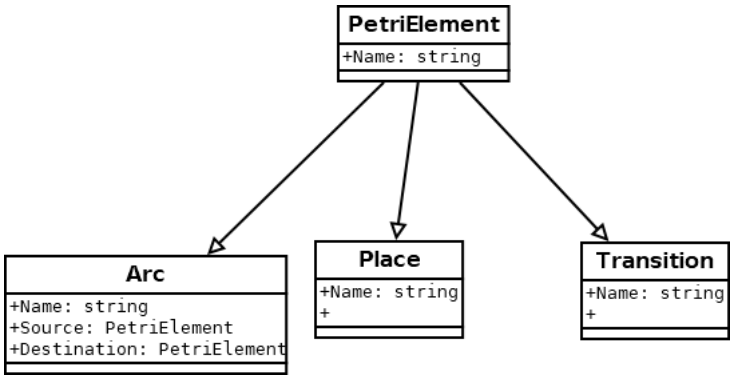
Az így megtervezett sablon a 4.1-es ábrán látható.



4.1. ábra. Az eddig leírt sablon

4.2.2. Petri háló

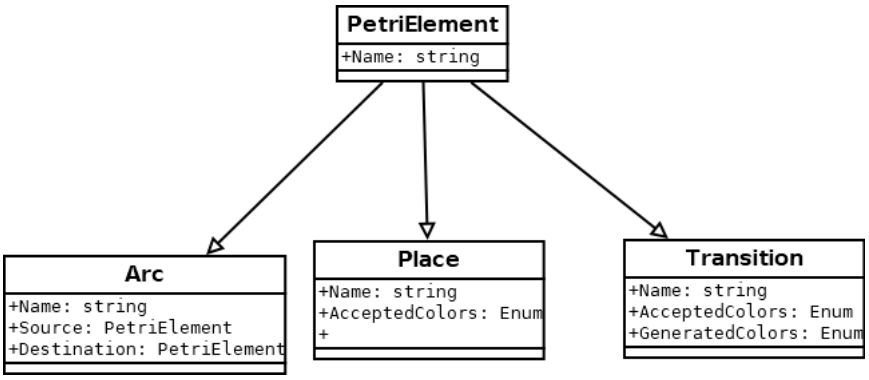
A következő lépés a tervezetben, a "végtermék" leírása. Ahogy a BPEL-nél az volt szem előtt, hogy egy XML file miként dolgozható fel, és milyen elemekből áll, itt az lesz fókuszban, hogy a hálónak két típusa van, de mindkettő ugyan azon két főelemből épül fel, helyekből és tranziciókból. Alapvetően egy hasonló sablont kapunk itt is, amelyet a 4.2-es ábrán látunk.



4.2. ábra. A petri hálók prototípusa

Észrevehető, hogy a két ábra nagyon hasonló, de az elsőn jóval több leszármazott osztályt kellene feltüntetni. Észrevehető, hogy a diagram nem tartalmaz semmit, ami a színezett variánst megkülönböztetné. Ha megfigyeljük a színezett hálók leírását, rájövünk, hogy elég csak a hely és tranzició osztályt kibővíteni. Az így bővített osztályt a 4.3 ábra mutatja. Az élek gyakorlatilag csak a vizualizáció miatt szükségesek, ugyanis, az előbb említett osztályokat ha kiegészítjük *mikor tüzelhet, milyen színnel, hová tüzelhet, stb.* paraméterekkel, akkor működésében a háló nem változik, de innentől a rajzolandó ábra nem tekinthető Petri hálónak, hisz nincs benne él. Viszont az előbbi észrevétel hasznos a szimuláció és megjelenítés során, ugyanis nem történik mozgás az élen belül, csak a két végpontja között.

Az így kapott ábra alapján már le tudjuk írni a színezett hálókat is. Észrevehetjük, hogy a helyek, az ábra szerint nem generálnak tokent. Viszont előfordulhat, hogy kezdéskor, vagy felhasználói bemenetre egy adott helyen jöhet létre token. Ez viszont nem mond ellent az ábrának, csak látszólag, ugyanis a helyek tényleg nem generálnak tokent, hanem inicializálva lesznek az adott mennyiségű tokennel, illetve felhasználói bevitel esetén a token a helyen megjelenik, kvázi mintha egy "rejtett" tranzició által



4.3. ábra. A színes háló

kerülne oda a token.

4.3. A vizualizáció megtervezése

A vizualizáció gyakorlatilag két részből áll:

- a számítás eredményeinek listázásából,
- a háló megjelenítéséből.

Az eredmények listázása nem számít különösebben nagyobb feladatnak, ugyanis a kiíratást egyszerű szövegdobozok vagy label-ek elvégzik. Ami a vizualizációnál érdekesebb folyamat, az a háló generálása. A háló generálásánál, két opció adódik: az egyik, hogy sajt komponenst írunk a feladatra, a másik, hogy olyat használunk, ami már meg van írva és elérhető (lehetőleg szabadon). Utóbbi esetben szükséges egy olyan gráfleíró-nyelv használata, amit a program el tud fogadni. Az utóbbi opció sokkal logikusabb több indok miatt is:

- az open source szoftverek általában jól optimalizáltak,
- kevés helyet foglalnak,
- időt lehet vele megtakarítani, és
- többnyire kielégítő dokumentációja van.

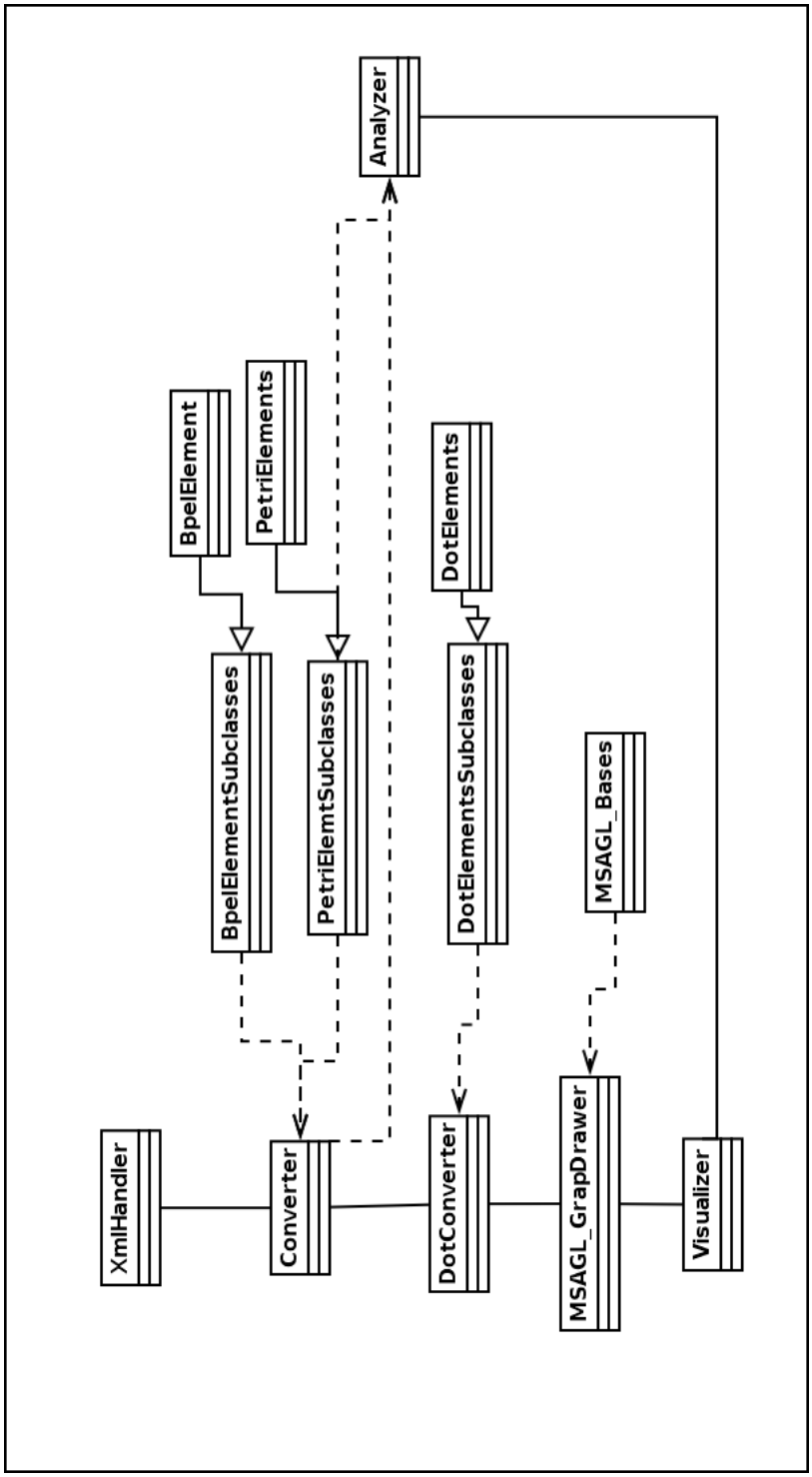
Innentől kezdve ha eldőlt a grafikus megjelenítés miként, a következő lépés a hozzá szükséges gráf előállítása. Ez szintén két lépcsős folyamat. Először a forrásból kell Petri hálót generálni, majd a hálóból egy gráfleírónyelvi verziót. Ha tokenáramot szeretnénk rajta ábrázolni, akkor viszont olyan ábrát kell készíteni, ami könnyedén változtatható, és kellően látszik benne a tokenáram. A láthatóságot a gráfrajzoló legtöbbször önállóan vizsgálja, és úgy szerkeszti a gráfot, hogy az megfelelően olvasható legyen. A tokenáramot viszont kétféleképpen lehet ábrázolni: egy részeiben rajzolható aktív felületen, vagy egy teljesen passzív felületen, az egész kép cserélésével. Az első megoldás egy erre specializált architektúrát igényel, ami általában nem ingyenes, vagy nagyon sok módosítást igényel. A második módszer viszont szinte bárhol működik, cserébe a képét teljes egészében le kell generálni, így nagy háló esetén, illetve nagy számú tokenek esetén valószínűleg egy számításigényes időszakot kezd el.

4.4. A leképzések megtervezése

A szoftver egyik fő komponense a konverter. Ennek feladata a tényleges átalakítás előfeldolgozást követően. A konverternek tudnia kell egy adott elemből egy az elemnek megfelelő részhálót előállítani. A konverzió során figyelni kell az eredeti elem tulajdonságait. A tulajdonságok után be kell állítani az új node típusát. Tranzició ugyanis működhet 'AND' és 'OR' módban is. Bár mindkettőre leggyakrabban binér műveletként tekintünk, a hálóban egy tranzició viszont nem feltétlenül csak kettő bejövő élt tartalmaz. A konverzió során ügyelni kell az eredeti elem tulajdonságaira, például mik az előfeltételek, vagy esetleg az aktivitás milyen további tokenmozgást indíthat el, és ennek hatására a háló további részein beállítani a megfelelő éleket. Ugyanakkor nem mindig képezhető le egy elem úgy, hogy megfelelően illeszkedjen a hálóra. A háló alapfeltétele ugyanis, hogy tranziciót csak hely követhet, és helyet is csak tranzició (élek közrefogásával).
A leképzés során használt forrás file viszont tartalmaz, a konverzió számára fölösleges XML nyelvi elemeket is. Például `porttype`, ami a folyamat külső eléréséhez szükséges. Az ilyen, és ehhez hasonló nyelvi elemek zöme azért elhanyagolható, mert a hálók csak helyre tudnak tokeneket fogadni külsőleg, és ameddig az meg nem történik a háló kvázi önszervező rendű.
Színes hálóknál viszont problémákba ütközünk. Az elsődleges probléma a színes tokenek. Az egyszerű háló csak egyféle tokent tud kezelni, emiatt lényegesen egyszerűbb a leképzése, de csak olyan folyamatokat enged leképezni, ahol a vezérjelet kell szimulálni. Tehát, ha vezérjel mellet megjelenik például egy függőség valamilyen külső objektumra, akkor a háló mindenképpen színes lesz (legalább kettő színnel).

4.5. Teljes terv

Ha az eddig elkészült részegységeket összeillesztjük, egy elég jó alapvető képet kapunk a szükséges komponensekről. Ezt a 4.4 ábra mutatja. Természetesen, mint minden rendszerterv ez sem tökéletes, hiszen legtöbbször implementáláskor történnek változtatások, általában az irányelvek betartása, valamilyen kényelmi szempont, vagy az előredefiniált implementálási módszerek hiánya miatt. Természetesen ilyenkor utólag az ábrát bővíteni szokás, de az már nem számít rendszertervnek, hanem a rendszer leírása mellett szokott helyet kapni.



4.4. ábra. Az összesített diagram

5. fejezet

Az üzleti folyamatok elemeinek leképzése

5.1. A leképzés menete

A BPEL nyelv az egymással együttműködő modulok működését szimulálja. A BPEL modellben a folyamat lépéseit tevékenységnek *(activity)* nevezzük. A tevékenységek lehetnek elemiek és összetettek. Tipikus elemi tevékenységek:

- más funkció meghívása,
- üzenet küldés/fogadás,
- változók módosítása,
- várakozás, szinkronizálás.

Az elemi tevékenységekből összetett folyamatok építhetőek fel szekvencia, párhuzamosítás és ciklus elemek segítségével.

A BPEL szabvány tevékenység készletének bemutatásánál csak a legfontosabb elemekre térünk most ki. A szabvány részletei a http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html weboldalról érhetőek el.

A BPEL definíció egyértelműsíti a folyamat kezdetét és inicializálja is a megfelelő paraméterekkel. A Petri háló erre nem tér ki, ezért bevezetünk egy opcionális *START* tranziciót. Ennek feladata, a megfelelő tokenek legenerálása a folyamat indításához. A folyamat zárása szintén egyértelmű a BPEL szabványban. Petri háló kapcsán beszélhetünk a háló leállásáról, ha már semmilyen hely és tranzició nem produkál új tokent, nem várakozik és nem nyel el tokent. A köztes elemek leképzése általánosan nem bonyolult, de megvizsgálhatók alternatív leképzési módok.

Az alkalmazás jelölésrendszere: A rendszer a helyeket P-vel, míg a tranziciókat T-vel indexeli. A helyek után listázza az adott helyen levő tokeneket. A jelenleg lépésben levő tranziciók pedig színes kitöltést kapnak. Alább egy pár, esetenként nem triviális elem leképzése látható, a többi elem leképzése és magyarázata a mellékletek között található.

5.2. <invoke>

Egy BPEL vagy épp egy webszolgáltatás meghívására szolgál és definiálja a szolgáltatás feladatát is.

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?>
   <catch faultName="QName"? ... >*
   activity
   </catch>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableNm"/>+
   </toParts>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableNm"/>+
   </fromParts>
</invoke>
```

A grafikus megjelenítése az 5.1. ábrán látható.

5.1. ábra. Az `invoke` grafikus jelölése az Oracle BPEL designer-ben

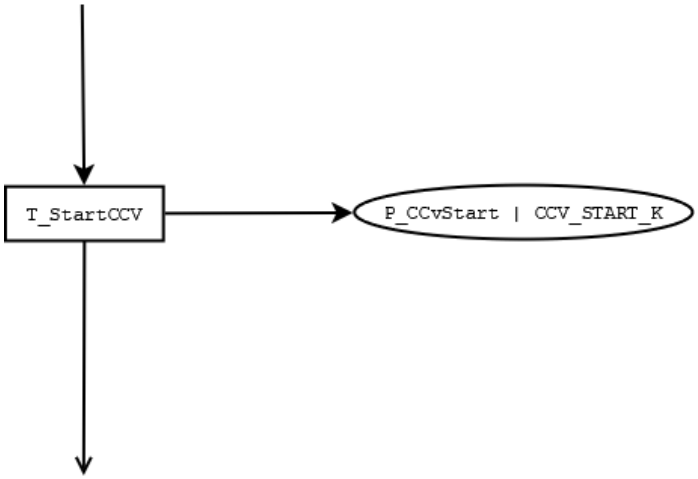
Használatát a programrészek újrafelhasználhatósága indokolja, valamint az átláthatósági alapelvek. Például, az ábrán látható CCvalidation használható ATM-es pénz felvét, egyenleglekérdezés, vagy egyéb ATM nél végezhető művelet során.

Leképzés során ügyelni kell arra, hogy az `invoke` paramétereinek megfelelő tokenek keletkezzenek és legyenek átadva a részháló start elemének.

5.3. <assign>

Egy változó értékadására szolgáló esemény. Ellentétben egy imperatív értékadással egy `<assign>` blokkban bármennyi értékadás, másolás történhet, amíg azt a kliens kezelni tudja, így logikailag egy egységbe zárja a műveleteket.

```
<assign validate="yes|no"? standard-attributes>
   (
   <copy keepSrcElementName="yes|no"?
   from-spec
   to-spec
```

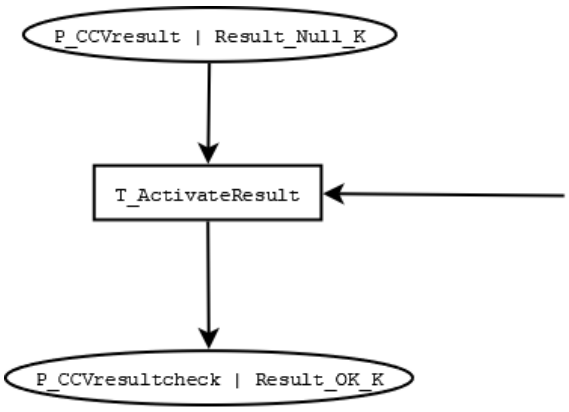


5.2. ábra. Az <invoke> leképzése petri hálóra

```
    </copy>
</assign>
```

Az érték hozzárendelése nagyon egyszerűen átírható egy tranzicióra ami a megfelelő tokenek színét módosítja. A grafikus megjelenítése az 5.3. ábrán látható. A színmódosítás egyszerűen a token nevének átírását jelenti.

5.3. ábra. Az assign grafikus jelölése az Oracle BPEL designer-ben



5.4. ábra. Az assign leképzése petri hálóra

Az ábrán egy változós értékadásra látható példa. Hasonló módon kezelendő a blokkosított értékadás, a különbség, hogy a több token jöhet és távozhat több forrásba is.

5.4. <validate>

Egy sémára validálja az XML (BPEL) állományt.

```
<validate variables="BPELVariableNames" standard-attributes>
   standard-elements
</validate>
```

Mivel a Petri háló nem tartalmaz validációs elemeket, ezért a <validate> nem képződik le.

5.5. <throw>

Egy rész processzen belül fault generálására szolgál.

```
<throw faultName="QName"
   faultVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
</throw>
```

Nagyon egyszerűen egy *fault* tokent generáló tranzició komponens. Explicit hálórésze nincs, hanem a megfelelő inputtokenek megléte vagy hiánya generálja egy tranzició során.

5.6. <wait>

Időre vonatkoztatva várakoztat. Például 5000 tick vagy 14:00:23 (hh:mm:ss)

```
<wait standard-attributes>
   standard-elements
   (
   <for expressionLanguage="anyURI"?>duration-expr</for>
   |
   <until expressionLanguage="anyURI"?>deadline-expr</until>
   )
</wait>
```

Megadható egy részhálóval ami valójában egy oszcillátor, és a megfelelő iteráció után *continue* tokent küld. Továbbá létezik úgynevezett időérzékeny háló, ahol az elemek tokenátadási sebessége ismert, vagy állítható. Ezzel időzítőt lehet létrehozni. Esetenként hardware-es óra is használható.

5.7. <empty>

No-op (*no operations*) esemény szinkronizációra szolgál.

```
<empty standard-attributes>
   standard-elements
</empty>
```

Beiktatható egy semleges tranzició és hely.

5.8. <sequence>

Sorozatot ad meg.

```
<sequence standard-attributes>
   standard-elements
   activity+
</sequence>
```

Egyszerűen csak tranziciók és helyek összefűzése.

5.9. <if>

<if> Standard kétirányú elágazás. Logikai XPATH kifejezést vár.

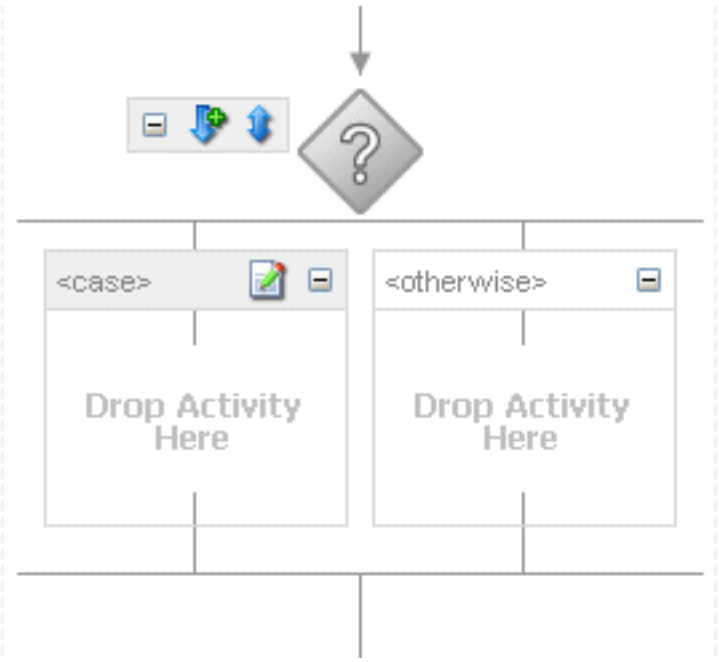
```
<if standard-attributes>
   standard-elements
   <condition >bool-expr</condition>
   activity
   <elseif>*
      <condition>bool-expr</condition>
   activity
   </elseif>
   <else>?
   activity
   </else>
</if>
```

Egy tranzició, mely tokenek függvényében más felé küldi tovább, vagy generál tokeneket. Ez lehet egy helyen összegyűlt tokenek mennyisége, vagy egy adott helyen egy specifikus színű token megléte, vagy nemléte. Analóg módon egy *Switch-Case* elágazás is definiálható vele. A PBEL-es grafikus megjelenítése az 5.5. ábrán látható.

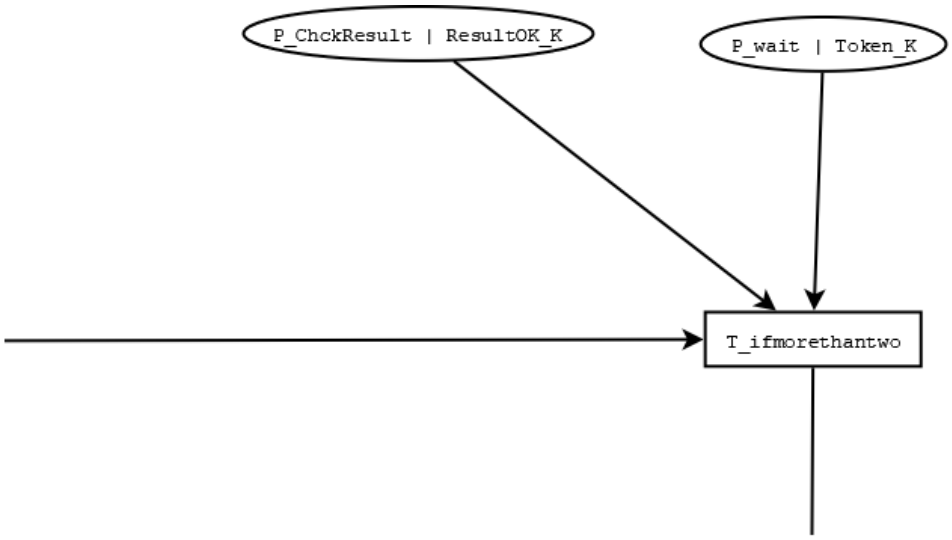
5.10. <while>

Elöltesztelő ciklus. Addig iterál, amíg az ciklus feltétel igaznak értékelődik ki.

```
<while standard-attributes>
   <condition>bool-expr</condition>
   activity
</while>
```



5.5. ábra. Az `if` grafikus jelölése az Oracle BPEL designer-ben



5.6. ábra. Az `if` Egy hálóképzése

Egy tranzició, mely token függvényében a folyamat egy korábbi pontjára csatol vissza, vagy éppen egy későbbire, a feltétel hamis logikai állapota esetén. A feltétel persze egy színes token jelenléte, vagy tokenek száma is lehet.

5.11. <repeatUntil>

Egy hátultesztelős ciklusnak feleltethető meg, amely akkor enged tovább, ha a feltétel igaz.

```
<repeatUntil standard-attributes>
   standard-elements
```

```
   activity
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
</repeatUntil>
```

A <while>-al analóg módon megadható a Petri hálós leképzése.

5.12. <forEach>

Végig iterál a gyerekelemeken. Megadható párhuzamos feldolgozás is. Egy *Complete condition* segítségével megadható egy break utasítás, ami kilép a ciklusból.

```
<forEach counterName="BPELVariableName" parallel="yes|no">
   <startCounterValue expressionLanguage="anyURI"?>
   unsigned-integer-expression
   </startCounterValue>
   <finalCounterValue expressionLanguage="anyURI"?>
   unsigned-integer-expression
   </finalCounterValue>
</forEach>
```

Egyszerű loop utasítás, azonban párhuzamosítás esetén a részhálóból megfelelő példányszámot generáltatunk.

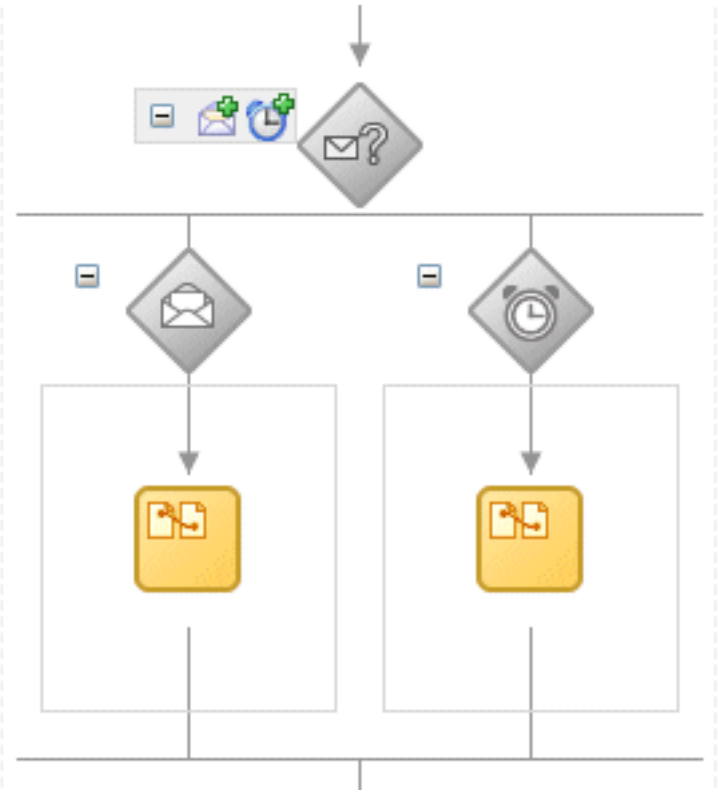
5.13. <pick>

Üzenetek várására vagy időtúllépés eseményre figyel. Ezek bármelyike a szubprocessz végrehajtásához vezet.

```
<pick createInstance="yes|no"? standard-attributes>
   standard-elements
   <onMessage partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      variable="BPELVariableName"?
      messageExchange="NCName"?>+
      <correlations>?
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      activity
   </onMessage>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )
```

```
      activity
   </onAlarm>
</pick>
```

A grafikus megjelenítése az 5.7. ábrán látható.



5.7. ábra. A `pick` grafikus jelölése az Oracle BPEL designer-ben

Összetartó hálóval és egy tranzicióval képezhető le.

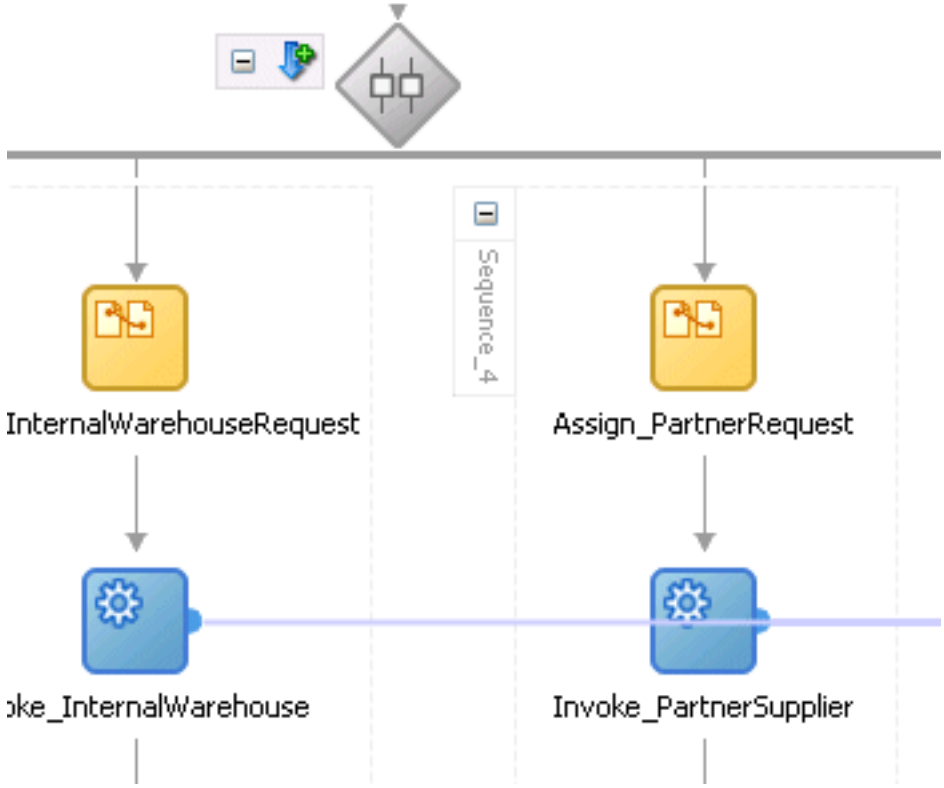
5.14. <flow>

Konkurens elemek deklarálására szolgál. Linkek segítségével megadható függőségi viszony a gyerekek között.

```
<flow standard-attributes>
   standard-elements
   <links>?
      <link name="NCName" />+
   </links>
   activity+
</flow>
```

A grafikus megjelenítése az 5.8. ábrán látható.

Leképzése egyszerű, hisz a megfelelő gyerek elemek nem szekvenciálisan helyezkednek el, hanem párhuzamosan.



5.8. ábra. A `flow` grafikus jelölése az Oracle BPEL designer-ben

Az idáig említett leképzések az egyszerű hálóra lettek bemutatva. Színes háló esetén már sokkal összetettebb a folyamat. Ekkor ugyanis, mivel a színválasztás nem szigorúan meghatározott, a háló módosulhat a színválasztás függvényében. A legcélravezetőbb színválasztás viszont, az alap folyamat egy adott helyén előforduló item-ek más helyen történő előfordulását. Például acélgyártásnál minden olyan folyamat, ahol daruzható csille szükséges, tartalmazni fogja a "csille" színű tokent a hálóban. A színezéskor előfordulhat kettő szélsőséges eset, melyek:

- a háló minden pontja ugyan olyan színű, (ekkor a háló gyakorlatilag egyszerű háló),
- a háló minden pontja különböző színű, ilyenkor minden node-ra (hely és tranzició) ki kell számolni a szükséges értékeket (ugyanis nincs "átadó" folyamat, ahol az input megegezik az outputtal).

6. fejezet

A hálón végezhető elemzések

6.1. Háló korlátosság és puffer kapacitási ellenőrzés

A háló egy adott helye akkor tekinthető korlátos (*bounded*) helynek, ha bármely jelölésnél a tokenek száma az adott helynél nem megy egy adott korlát fölé. A Petri háló korlátos, ha minden helye korlátos hely. A háló korlátossága az egyik leggyakoribb és legfontosabb minőségi jellemzője a Petri hálóknak.

A háló alap tulajdonságainak, beleértve a korlátosságának az elemzésére több módszer is létezik, melyek közül kiemelhető a

- komponens/elérhetőségi gráf elemzése (SCC),
- dekompozíciós módszerek.

A mátrix reprezentáció esetén transzformációs mátrixok segítségével írják fel a Petri háló dinamikáját. Az alapstruktúra az ú.n. incidencia M mátrixban kerül megadásra, melynek elemei az alábbi jelentéssel bírnak: $A_{ij} = a_{ij}^{+} - a_{ij}^{-}$, ahol

- a_{ij}^{+} : az élerősség az i. tranzicióból a j. kimeneti hely felé,
- a_{ij}^{-} az élerősség az i. tranzicióhoz a j. bemeneti hely felől.

A mátrix alapvetően a tokenek számának a változását mutatja az egyes tranzició átmenetek esetére. A Petri háló működési alapegyenlete a következő alakban adható meg:

$$M_k = M_{k-1} + Au_k,$$

ahol M_k jelöli a háló markereinek (tokenek) státuszát a k. lépésben. Az u vektor a helyek tüzelési státuszt írja le.

A fenti modellen alapuló elérhetőség vizsgálatok felhasználhatóak a korlátosság elemzésére [14]. A kapcsolódó egyenletek hatékony, lineáris programozási megoldása szintén ismert [12].

6.2. Saját modell alaphálóra

Modellünkben korlátosságnak egy folyam-gráf megközelítését dolgoztuk ki. A hálóban a következő típusú helyeket definiáljuk:

- forrás hely,

- nyelő hely,
- köztes hely.

Feltesszük, hogy csak a köztes helyeken lehet tokeneket tárolni, csak ott vannak pufferek. A hálóban az élekhez egy I_x token áramlás erősséget definiálunk, ahol x jelöli az él indexét. A forrás helyekhez egy Q_x forrás erősség indexet adunk meg. A hálóban az alábbi kapacitás korlátokat vezetjük be:

- C_x : az x. tranzició maximális erőssége,
- C_y : az y. nyelő maximális folyam erőssége.

A tranzicióknál a bemenő élek vonatkozásában kétféle működési módot értelmezünk:

- AND-mód: akkor van tüzelés, ha minden bejövő élnél megvan az al elvárt tokenszám, van szinkron,
- OR-mód: akkor van tüzelés, ha megjelenik valamely bemeneten egy token, nincs szinkron.

A hálóban az alábbi megkötések élnek a folyamerősségekre:

- forrás helyek esetén: $\sum_y I_y = Q_x$ (y: kimenő élek),
- nyelő helyek esetén: $\sum_y I_y \leq C_x$ (y: bejövő élek),
- belső helyek esetén: $\sum_{y(ki)} I_y \leq \sum_{y(be)} I_y$,
- tranziciók esetén:
  - $\sum_y I_y \leq C_x$ (y : bejövő élek),
  - $\sum_{y(ki)} I_y = \sum_{y(be)} I_y$,
  - $\forall$ kimenő x, y élre $I_x = I_y$.

Az AND típusú tranzakciók esetén még ezen felül teljesül, hogy $\forall$ bejövő x, y élre: $I_x = I_y$.

Az egyes belső helyeken a pufferbe áramló tokenek eredő intentitása:

$$F = \sum_{x(\text{belső hely})} \left(\sum_{y(\text{x bejövő él})} I_y - \sum_{y(\text{x kimenő él})} I_y \right)$$

Az F függvény 0 értéke esetén nincs szükség belső pufferre.

A fenti feladat egy LP programozási feladatnak is tekinthető, ahol a változók az élek I_x nem negatív intenzitásai, és a célfüggvény:

$$F \to \min$$

alakú.

6.3. Az alkalmazott, kibővített modell színezett hálóra

A színezett Petri hálók esetén több különböző típusú tokenek élnek a rendszerben. A kapacitás vizsgálatnál ekkor az egyes tranzicióknál eltérő lehet a kapacitás korlát (a maximális folyam erősség) a különböző típusú tokenek esetén. Emiatt külön kell vizsgálni az egyes típusok folyam erősségét, nem lehet összevonni őket.

A színezett hálóban az élekhez I_x^c token áramlás erősségeket definiálunk, ahol x jelöli az él indexét és c a színkód. A forrás helyekhez Q_x^c forrás erősség indexeket adunk meg a különböző c színekre vonatkozólag. A hálóban az alábbi kapacitás korlátokat vezetjük be:

- C_x^C: az x. tranzició maximális erőssége a c szín esetén
- C_y^C: az y. nyelő maximális folyam erőssége a c szín esetén

A hálóban az alábbi megkötések élnek a folyamerősségekre:

- forrás helyek (x) esetén: $\forall c$ színre: $\sum_{y \text{ kimenő élek}} I_Y^C = Q_x^C$,
- nyelő helyek (x) esetén: $\forall c$ színre: $\sum_{y \text{ bejövő élek}} I_y^C \leq C_x^C$,
- belső helyek esetén: $\forall c$ színre: $\sum_{y \text{ kimenő élek}} I_y^C \leq \sum_{y \text{ bejövő élek}}$,
- tranziciók (x) esetén:

$$\forall c \text{ színre} \sum_{y \text{ kimenő élek}} I_y^C == \sum_{y \text{ bejövő élek}} I_y^C$$

,

$$\sum_{c \text{ színek}} \left(\frac{1}{C_x^C} \left(\sum_{y \text{ bejövő élek}} I_y^C \right) \right) \leq 1$$

,

$$\forall c \text{ színre:} \forall \text{ kimenő} (y,z) \text{élre: } I_y^C = I_z^C$$

,

- az AND típusú tranzakciók esetén még ezen felül teljesül, hogy $\forall c$ színre: $\forall$ bejövő (y,z) élre $I_y^C = I_z^C$.

Az egyes belső helyeken a bufferbe áramló tokenek eredő intenzitása:

$$F = \sum_{c \text{ színek}} \left(\sum_{x \text{ belső hely}} \left(\sum_{y \text{ bejövő élek } x\text{-nél}} I_y^C - \sum_{y \text{ kimenő helyek } x\text{-nél}} I_y^C \right) \right).$$

Az F függvény 0 értéke esetén nincs szükség belső bufferre. A fenti feladat egy lineáris programozási feladatnak (röviden LP) is tekinthető, ahol a változók az élek I_x nem negatív intenzitásai és a célfüggvény $F \to \min$ alakú.

6.4. A validációs számítás algoritmusa

A hálót leíró struktúra három alappilléren nyugszik: helyek, tranziciók, élek.

A helyek esetén az alábbi attribútumokat tárolja a rendszer:

- `id`: az egyedi azonosító kód,
- `inputs`: bejövő élek,
- `outputs`: kimenő élek,
- `tokens`: tárolt tokenek,
- `Q`: forrás intenzitás,
- `border`: pozíció jelző, belső vagy határ pozíció.

A tranzíciók jellemzői:

- `id`: egyedi azonosító kód,
- `inputs`: bejövő élek,
- `outputs`: kimenő élek,
- `C`: feldolgozási intenzitás,
- `mode`: működési mód (AND, OR).

Az élek attribútumai:

- `id`: azonosító kód,
- `input`: induló elem,
- `output`: cél elem,
- `alfa`: az él kapacitás jelzője,
- `inner`: él típusa, belső vagy határ.

A program python szegmense később integrálásra került a fő programban. Különböző változásuk után a Goole OR tools lett a végső részmodul, amit erre a célra alkalmaztam. A szintaktikája hasonló a pythonéhoz, de helyenként különbözik. Az előző példa C# ban a GLOP segítségével megoldva:

A sover létrehozása GLOP-al

```
Solver solver = Solver.CreateSolver("GLOP");
```

A változókat itt megadhatjuk egy előre definiált tömbbel, vagy egyesével. A programban konverzió miatt az utóbbi lett kiválasztva, mert folyamatosan feltöltető.

```
Variable weights = solver.MakeNumVar(0.0, int.PositiveInfinity, "weight");
```

Megkötés hozzáadása: példa belső élre

```
solver.Add(c[2]—c[5]>0);
```

A célfüggvény:

```
solver.Minimize(i => costs[i] * trVars[i]);
```

Ha kész az előfeltétel, akkor elindítjuk a solvert

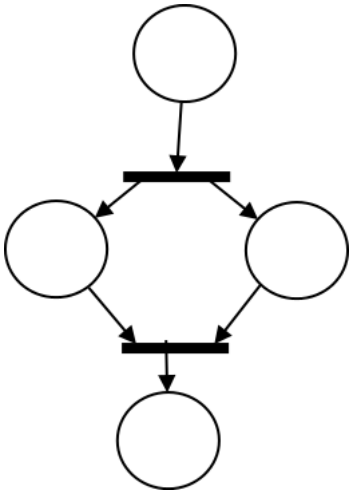
```
Solver.ResultStatus resultStatus = solver.Solve();
```

Itt a result status tartalmazza a megoldást, ha van, ekkor 'OPTIMAL'-al tér vissza, a változók ekkor kinyerhetőek:

```
sb.Add(c[1].SolutionValue());
```

6.5. Mintafeladat

Vegyünk egy 4 helyből és 2 tranzicióból álló rendszert. A helyekből egy nyelő, egy forrás és kettő belső hely. A rendszerben 6 él van az ábrán megadott módon. 6.1 A gráfban a kerek elem a helyeket, szögletes a tranzíciókat jelöli. A csomópont elemben az első jel a hely kódja, a második a kapcsolódó kapacitás érték. A modellben a kisebb indexű tranzició AND tulajdonságú, a másik OR tulajdonságú.



6.1. ábra. A mintahálónk

A rendszerben 6 (nem negatív) változó jelenik meg:
$\{0 : Iv_0, 1 : Iv_1, 2 : Iv_2, 3 : Iv_3, 4 : Iv_4, 5 : Iv_5\}$
Az élek indexelése:

$$0 : 1 \to 5$$
$$1 : 5 \to 2$$
$$2 : 2 \to 6$$
$$3 : 6 \to 3$$
$$4 : 3 \to 5$$
$$5 : 6 \to 4$$

A rendszerhez az alábbi egyenlőtlenségek kapcsolódnak:

$$\begin{aligned}
C_1 &: Iv_0 &&= 20 \\
C_2 &: Iv_1 - Iv_2 &&\geq 0 \\
C_3 &: Iv_3 - Iv_4 &&\geq 0 \\
C_4 &: Iv_5 &&\leq 25 \\
C_5 &: Iv_0 - Iv_1 + Iv_4 &&= 0 \\
C_6 &: Iv_0 + Iv_4 &&\leq 50 \\
C_7 &: Iv_0 - Iv_4 &&= 0 \\
C_8 &: Iv_2 - Iv_3 - Iv_5 &&= 0 \\
C_9 &: Iv_2 &&\leq 50 \\
C_{10} &: Iv_3 - Iv_5 &&= 0
\end{aligned}$$

A kapcsolódó célfüggvény:

$$1 \cdot Iv_1 + -1 \cdot Iv_2 + 1 \cdot Iv_3 + -1 \cdot Iv_4 \rightarrow \min$$

.

Az LP feladat megoldható és a kapott megoldás:

$$\begin{aligned}
Iv_0 &: 20.0 \\
Iv_1 &: 40.0 \\
Iv_2 &: 40.0 \\
Iv_3 &: 20.0 \\
Iv_4 &: 20.0 \\
Iv_5 &: 20.0 \\
Cost &= 0.0
\end{aligned}$$

Tehát a mintarendszerben nincs szükség belső pufferre. Ha lecsökkentjük a második tranzició folyamerősségét, az akkor nem kapunk érvényes megoldást.

$$\begin{aligned}
Iv_0 &: 10.0 \\
Iv_1 &: 20.0 \\
Iv_2 &: 20.0 \\
Iv_3 &: 10.0 \\
Iv_4 &: 10.0 \\
Iv_5 &: 10.0
\end{aligned}$$

7. fejezet

Megvalósítás

7.1. Kezdetek

Még a téma megfogalmazása előtt eldöntöttem, hogy a szakdolgozatomhoz tartozó programot C# nyelven fogom írni. Ennek legfőbb oka az eddig megszerzett nyelvi jártasságom. Magát a nyelvi verziót és a program alatt futó keretrendszert viszont a feladathoz igazítottam. Először a .NET Core 5 lett kiválasztva de kényelmi okokból, illetve mivel a platformfüggetlenség nem volt elsődleges szempont ezért a .NET 4.7.2-es verziója lett a projekt keretrendszere. A NuGet a C# csomagkezelője. Már megírt modulok találhatóak rajta, melyek között van ingyenes és licenszelhető is. Egy érdekes tapasztalat, hogy a NuGet csomagokat legtöbbször nem sikerül a fejlesztőknek egy éven belül frissíteni, ezért sokszor nem célszerű az éppen legfrissebb .NET verziót használni. (A kívánt csomagok frissülése után a projekt is .NET 4.8-ra lett migrálva.) A C# alatt elérhető leggyakrabban használt fő alkalmazástípusok:

- ASP Web projekt és a hozzá választható front-end,
- Windows Console alkalmazás (Főként utility projektekhez használják),
- Windows Forms "klasszikus" ablakos alkalmazás gombokkal és textboxokkal. (Leginkább a kétezres évek formavilága),
- WPF (Windows Presentation Forms) Az előző egy "felújított" változata. Ez lényegesen több opciót biztosít a felhasználói felület testreszabására.

A projektválasztás a Windows Forms alkalmazástípusra esett (továbbiakban WF). A WF ugyan csak egyszerű grafikai megjelenítést tesz lehetővé, de egy gráf, vagy háló folyamatai könnyedén megjeleníthetőek. A WF cserébe könnyebben szerkeszthető, mint egy WPF XAML nyelven leírt felülete, és kevesebb erőforrást is igényel.

7.2. Fejlesztői környezet

7.2.1. Rövid történeti háttér

A nyelvválasztás részben együtt jár a fejlesztőkörnyezet választásával is. A legelterjedtebb nyelvek mellé társul legalább egy, kimondottan a nyelvhez fejlesztett fejlesztői környezet (IDE) is. A C# -hoz ajánlott és tervezett IDE a Visual Studio. Ennek legfrissebb kiadása a 2019-es. Eredetileg a Microsoft az általa kiadott nyelvekhez külön-külön fejlesztőkörnyezetet hozott létre, ám hamar kiderült, ez mind gazdaságilag, mind befektetett időben nem kifizetődő. Ezért először házon belül elkezdtek egy kombinált IDE fejlesztését, majd greenlight után kiadásra is áttervezték. Ez lett a Visual Studio 97. A '97-es verzió a C++, J++ és VisualBasic nyelveket támogatta natívan. A következő kiadás megtörte a később visszavezetett évszám rendszerű elnevezést és "csak" 6.0-lett. A kiadás mérföldkő volt, ugyanis ekkor jelent meg a .NET platform, aminek célja az akkor még hosszútávra tervezett két Windows rendszert, a 9x-et és az NT-t. A következő kiadás a Visual Studio .NET 2002 lett. a következő kiadással együtt (2003) léteztek különálló formában és Visual Studio 6.0 alá tölthető modulként is. (Így nem kellett külön telepíteni egy új IDE-t és újra beállítani egy auto-deploy rendszert.) A 2002-es kiadás mutatta be a C# nyelvet a nagyközönség számára. A C# lényegében az akkor a SUN kezében lévő JAVA nyelv helyetteseként született. A SUN ugyanis határozatlan időre felfüggesztette a JAVA és a hozzá tartozó virtuális gép Internet Explorer alatti támogathatóságát. Eddig ugyanis, a Visual Studio a JAVA egy implementációját támogatta: a J++-t. A Microsoft a J++-t két programnyelvvel helyettesítette: a J# -al és a C# -al. Előbbi elég rövid életűnek bizonyult, utóbbi viszont még létezik, sőt igen elterjedtnek mondható. A C# -ot a .NET el együtt fejlesztik, hiszen kimondottan ahhoz készült. Jelenleg a hagyományos .NET 4.8-as verziója a legfrissebb, de van platformfüggetlen .NET is, a .NET Core, jelenlegi nevén és kiadásán: .NET 5.

7.2.2. Általános bemutatás

A Visual Studio jelenleg három kiadása érhető el

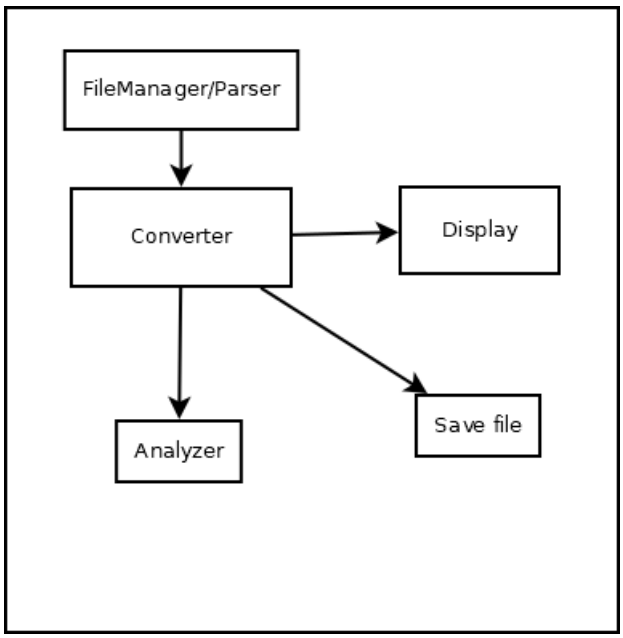
- Community,
- Professional,
- Enterprise.

A kiadások kimerítő összehasonlítása a Visual Studio oldalán érhető el[3]. Lényegében a Community egy ingyenes verzió, csökkentett funkcionalitással, a Professional, az analóg nevű kiadásnak felel meg, ami már teljes funkcionalitást biztosít, míg az Enterprise tartalmaz olyan funkciókat, melyek nagy, összetett projektek fejlesztését könnyítik meg. Jelenleg a Visual Studio nem tartalmaz minden támogatott nyelvet gyárilag, hanem a használni kívánt modulokat telepítéskor kell megadni. Így elkerülve a fölösleges helyfoglalást.
Az alapfunkciók közé tartozik az editor, a debugger, illetve a verziókezelés támogatása. Az általam használt Enterprise kiadás viszont ettől jóval többet tud. A kiadásra régebben volt szükségem, viszont öröklicenszes, ezért megmaradt. Sok kényelmi funkcióval látták el. Ilyenek például az élő függőség ellenőrzés, architektúra validáció, diagram szerkesztő, stb. Az általam egyik legtöbbet használt komponens a live debugging, ami

futás közben engedi a program szerkesztését és a live unit testing, ami kód írása közben ellenőrzi a unit tesztek eredményét, és a kód tesztlefedettségét. Debug közben sokat segít a valós idejű memória térkép is. Itt megtekinthető a program memóriahasználata, ami lebontható a különböző modulok memória igényeire is. Továbbá, a historical és snapshot debugging engedi az egy debug szálon nem előfordult, lehetséges scenariok memória, és CPU igényének megtekintését, és a különböző esetek egymás melletti futtatását egy debug session alatt.

7.3. Felépítés

A fejezet elején elmondottak alapján a projektet csak 4.7.2-es .NET alatt lehetett létrehozni. A projekt eredetileg egy TDK-hoz készült, később viszont módosítva lett. Az alkalmazás felépítése a következő:7.1



7.1. ábra. Az alkalmazás vázlatos felépítése

FileManager / Parser Feladata az XML file beolvasása, és tagolása. Az így feldolgozott adat kerül átalakításra a program későbbi szakaszában.
Konverter A program fő része. Feladata a már tagolt adathalmaz átalakítása. Az átalakított adathalmaz DLV formátumba kerül, majd gráfleíró nyelvre (DOT). Az így előállított háló menthető egy file-ba.
Display Megjelenítő modul a konvertált háló UI-ra rajzolásáért felelős. Kis háló esetén az animáció is lehetséges.
Analyzer A hálón végezhető elemzéseket végzi el. Majd ezek eredményét megjeleníti, illetve elmenti.

7.3.1. FileManager / Parser

A File manager a C# hagyományos XML megoldásait használja, ez az **XmlDocument** és **XmlReader** osztály illetve a hozzá tartozó metódusok. A felhasználói felületen a felhasználó által kiválasztott file kerül a managerbe. A C# támogatja az XML fileok

hagyományos XPath alapú feldolgozását, azonban elérhető egy gyorsabb XML feldolgozás LINQ-val. A LINQ a C# nyelvbe integrált lekérdező nyelv. Szintaktikája az SQL-éhez hasonló annyi különbséggel, hogy a 'SELECT' a query végére kerül az eleje helyett. Ez főként a kódban előforduló lekérdezések elkülönülése miatt lett így tervezve. A LINQ része továbbá a Lambda expression parser is, ami hasonló lekérdezéseket tud generáli, illetve feldolgozni. A forrásfile beolvasása és feldolgozása kicsit bonyolultabb, mint egy adat XML, hiszen az adat XML nem lehet tetszőleges mély. A BPEL ezzel ellentétben elvileg igen, viszont az ilyen file olvasása több szempontból is nehézkes. Egyrészt rekurziót kell alkalmazni, ami ha nem vigyázunk nagyon hamar tele szemeteli a memóriát, illetve okozhat futásidejű hibákat is, ha például nem olyan elem van a legmélyebb szinten, mint amit vár, például sima szöveg valamilyen element helyett. BPEL során annyi könnyítést meg lehet tenni, hogy a tetszőleges mélységet előtte elhagyom, és invoke-okkal saját processen belül generálok egy új sequence-t. Az így kapott file maximum 3 szint mély lehet (a rootot leszámolva). Ezt a funkciót nem építettem bele a programba, hiszen akkor kétszer kéne végig menni ugyan azon a forrás file-on. Ez a programban a következő kép néz ki (a következő oldalon):

```
XmlDocument bpelSource = new XmlDocument();
          bpelSource.Load(openFileDialog.FileName);
          XmlElement root = bpelSource.DocumentElement;
          Process process = new Process();
          process.name = root.GetAttribute("name");
          List<Sequence> sequences = new List<Sequence>();
          var sequenceNodes = root.ChildNodes.Cast<XmlNode>().Where(x => x.Name.Contains("sequence")).ToList();
          foreach (XmlElement item in sequenceNodes)
          {
              var childs = item.ChildNodes;
              List<BpelElement> bpelElements = new List<BpelElement>();
              foreach (XmlElement child in childs)
              {
                  var bpelElement = new BpelElement();
                  if (child.HasChildNodes)
                  {
                      foreach (XmlElement innerChild in child.ChildNodes)
                      {
                          var innerElement = new BpelElement();
                          foreach (XmlAttribute attribute in innerChild.Attributes)
                          {
                              innerElement.attributes[attribute.Name] = attribute.Value;
                          }
                          bpelElement.ownBpelElements.Add(innerElement);
                      }
                  }
                  foreach (XmlAttribute attribute in child.Attributes)
                  {
                      bpelElement.attributes[attribute.Name] = attribute.Value;
```

```
    }
    if (!string.IsNullOrEmpty(child.InnerText))
    {
        bpelElement.attributes["innerText"] = child.InnerText;
    }
    bpelElements.Add(bpelElement);
}
sequences.Add(new Sequence(item.GetAttribute("name"),bpelElements));
}
process.bpelSequence = sequences;
```

7.3.2. Konverter

A Konverter a program fő modulja. A leképzés fejezetben felsorolt nyelvi elemeket dolgozza fel és készíti el a részhálókat, amiket aztán összefűz. Konverzió során az XML soronként kerül feldolgozásra. Egy átolvasás után a nem szorosan illeszkedő részek, mint részhálók, segéd folyamatok stb. külön szálak kerülnek feldolgozásra aszinkron módon. Ezzel a konverziós idő (a modelltől függően) rövidül. A konverzió gyakorlatilag egy keresés egy Dictionary-ben. A Dictionary egy kulcsot vár ami alapján egy értéket ad vissza. (Pont, mint egy valós szótár, ahol a kulcs az idegen nyelvű szó, az érték pedig a szó magyar megfelelője). Egy kisebb méretű Dictionary keresés durván $O(n)$, ha n darab elemre keresünk, ugyanis a `Dictionary.Contains()` egy elemre $O(1)$ komplexitású. Keresés után ezeket még rendezni kell és összefűzni. Ez utóbbi szintén minden elemre végbemegy, de vele egyidejűleg egy validáció is. A konverzió során az XML elem attribútumai és gyerekei alapján kerül az elem a háló egy adott pontjára. Ez egy újabb $O(n)$ lépés. Ha a rész szálak kész vannak és a fő szál is elkészült a részek egy hálóba lesznek rendezve ami $O(s)$, ahol az s a részhálók száma. Idáig a program tehát $O(2n + s)$ lépést tesz meg.

7.3.3. Analyzer

Az analyzer feladata az ötödik fejezetben leírtak gyakorlati alkalmazása az elkészült hálóra. Az analyzer eredetileg pythonban lett megírva. Ezt az indokolta, hogy az akkori alapalkalmazáshoz külön készült, mintegy kiegészítőként. Később a programba integrálva lett. Először IronPython alatt került bele, később teljesen átírva AGLIB-hez, majd még egyszer átírva jelenlegi Google OR toolset leírónyelvre. Az IronPython a python nyelv .NET-en futó implementációja. Gyakorlatilag kicsit gyorsabb a hagyományos pythonnál de cserébe a platform-függetlenséget feláldozza. (Jelenleg viszont az új .NET verzió futtatható Linux és MacOS rendszeren is.) Az AGLIB egy analitikus csomag aminek csökkentett képességű verziója elérhető a NuGet rendszerén keresztül ingyen. (A teljes verzió viszont licenszköteles.) A jelenleg használt Google OR tools egy ingyenesen elérhető open source software csomag optimalizáláshoz, LP, CP programozáshoz, illetve útválasztásos és flow feladatok megoldásához. A csomag előnye a felépítésében rejlik. A probléma leírása után egy proto állomány készül, amit egy solver fog megoldani. A solver LP problémáknál a GLOP, de külső solver is használható. A solverek implementáltak több platformra is. Az elérhetőség és a független problémaleírás eredménye, hogy az OR tools támogatott C++, C# , JAVA és Python nyelven is.

7.3.4. Kezdeti nehézségek

Miután körvonalazódott a feladat megvalósításának menete, létre hoztam egy kanban táblát a fejlesztési folyamatok nyomon követéséhez. (Ez a verziókezelővel együtt online, az Azure felületén érhető el.) A táblában különböző lebontásokban látható, a fő és rész egységek, valamint a feature-ök és tesztek. Az első lépés a BPEL nyelv megismerése volt. A BPEL-hez található anyagok nagy része hagyaték anyag, ugyanis a BPEL alapú folyamatkövetés kiszorult a népszerűségből. Ennek több velejárója van, de a legszembetűnőbb az aktívan elérhető segítség hiánya, a már hiányos dokumentáció, vagy éppen a dokumentációban felsorolt anyagok elérhetőségének hiánya. A program készítése során

kellett generálni egy úgymond tanító anyagot, amely a program részeit érthetően elmagyarázza, és későbbiekben lehet használni a konverzió és a feldolgozás helyességének ellenőrzésére. Ezen tesztadatok legenerálása sokkal nagyobb nehézséget jelentett, mint vártam, vagy mint a program többi részének egy hiba utáni javítása.
Ismerjük meg a generáláshoz szükséges feltételeket! A BPEL írására lehet egy sima szövegszerkesztőt alkalmazni, de ez nem célravezető, mivel a háttér XML validációja kézzel nehézkesen oldható meg. A leggyakrabban hivatkozott szerkesztő az ECLIPSE. Az IDE nem támogatja alapértelmezetten a BPEL szerkesztését, ezért egy beépülő szükséges. A rendszer egy URL megadása ellenében az adott szerveren tárolt beépülők letöltését, regisztrációját és integrálását teszi lehetővé. A folyamat több helyen is félre tud siklani, először az Eclipse verziójának különbözetében, aztán a szerver által alkalmazott biztonsági beállításokon, legutoljára az operációs rendszer jogosultságkezelésén. Sajnos a legfrissebb verziójú Eclipse nem támogatja a már idejét múlt szerkesztőt, ezért v6.0 körüli IDE-t kell telepíteni. A telepítés sajnos csak hagyaték rendszeren végezhető el (ez lehet mind Linux, mind Windows). A programnak Windows 2000/XP a rendszerkövetelménye hisz egy durván 2000-2002 es rendszerről beszélhetünk. Szerencsére a hagyaték rendszerek terén előzőleg szerzett tapasztalataim segítségemre voltak. A kettő Windows dual-boot egy korhű gépre telepítése után az editor hajlandó volt települni, egy azt megelőző gyakorlatilag verzió-hackelést követően. A frissebb Eclipse-el el kellett hitetni, hogy az állomány friss, de nem szabad volt feltelepíteni, hanem az így kinyert köteget kellett a másik gépen az Elcipse alá telepíteni manuálisan. Az IDE beállításai után lehetőség nyílt egy program szerkesztésére, de ekkor még csak validálatlan formában. Ekkor kiderült, hogy a szerkesztés befejezéséhez és validációjához egy szerverre van szükség. A szerver lehet local szerver is. A jelenleg Oracle kézben levő rendszer ehhez nem ad segítséget, de interneten található hozzá segédanyag [1]. Az ebből megismert Apache és ODE szerver szintén csak korhű rendszeren tud futni. A helyzetet bonyolítja, hogy a ODE-War állományokhoz JRE + JVM is szükséges. A JRE egy bizonyos verziója viszont inkompatibilis az ODE war állomány kifejezetten ahhoz tervezett verziójával. A JRE, apache tomcat, ODE verziók közösen nagyon kevés működő kombinációt engednek meg. Ennek ellenére, a telepítés befejeztével a legenerált minta még mindig nem ellenőrizhető, hiszen a szervernek futnia is kell. Ellenben a futó szerverbe fel kell regisztrálni az Oracle validációs szerverét, ahonnan egy aktuális sémát vár. A letöltött séma gyakorlatilag a BPEL verziót határozza meg. Sajnos ez a szerver 2010 óta nem működik, így a hosszadalmas munka után nem várt végeredményhez jutottam. Az így felállított rendszerrel 10 darab BPEL állományt generáltam, de mivel nagy része szerveresen állítható és kapcsolat nélkül helyfoglaló szöveg van kulcsfontosságú helyeken, így a minta felét el kellett dobni, mert szemmel láthatólag nem volt helyes. A mintagenerálást két hónap elteltével később újra megpróbáltam egy virtuális gépen, de ekkor még kevesebb eredmény született, ugyanis az Oracle ekkor kezdte a repository-k újraszervezését. Az egy hetes átszervezés után tapasztaltam, hogy a projektet nyugdíjazták, és jelenleg sem érhető el csak a hagyaték oldalon, de tényleges letölthető forrás nélkül. A dolgozat ezen fejezetének írásakor az oldal még mindig nem érhető el.

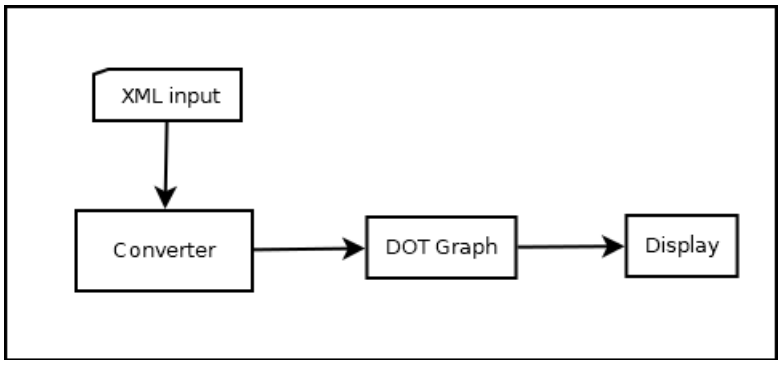
7.3.5. A konverter

A konverzió kezelését különösen problémásnak gondoltam. Tanulmányaim során sose volt említve összetettebb adattípusok feldolgozása és azok közötti konverzió. Először

egy kétszeresen összetett adattípusban gondolkoztam, ami gyakorlatilag egy véges, két dimenziós tömb, hiszen a BPEL csak véges számú elemet tartalmaz gyárilag, és a szabadon definiált elemekkel nem foglalkozunk. A tömb egyik dimenziója a bejövő BPEL elem, a másik pedig a hozzá tartozó előre meghatározott konverzió eredménye. Később eszembe jutott a szakmai gyakorlat alatt szerzett információ a C# -ban natívan jelen levő összetett adatstruktúráiról. Így esett a választás végül a Dictionary-re. A Dictionary egy 'Key - Value' azaz kulcs és érték alapon működő tár. Használata során egy kulcssort kell gyártani a rekordok számára, és a kulcshoz tartozó értéket felvinni. A Dictionary runtime, azaz futás idő alatt bővíthető ha szükséges, de mivel csak előre definiált elemeket használunk, ezért ez kihasználatlan marad. Konverzió során külön kell kezelni a színes és egyszerű Petri hálót. Az egyszerű hálós konverzió során sokat segített Christian Stahl által közzétett publikáció [9].

7.3.6. Vizualizáció

A vizualizáció az alábbi módon történik:



7.2. ábra. A vizualizáció menete

A konverter előállítja a hálót, ez ekkor még nem megjeleníthető petri háló struktúrában van. Ezt a struktúrát még át kell alakítani egy Gráfrajzoló program számára is érthető formátumba. A műveletet szintén a konverter végzi, ami a struktúrát tovább küldi az Analyzer számára, a gráfot pedig a rajzoló szubrutinnak. Az így kapott gráfot MS-AGL/AGLIB rajzolja ki a UI felületére. Ha a steb-by-step vizualizáció ki van kapcsolva, akkor a kirajzolt gráf a végleges. Ha viszont be van kapcsolva, akkor a tokenek fel kerülnek az ábrára és minden 1 lépés alatt lezajló művelet eredménye frissíti az adatstruktúrát. A struktúrából új kép készül, ami ki kerül a UI-ra. Ha a képeket szeretnénk elmenthetjük, ami a forrásfájl mappájába, vagy ha nincs jogosultság, akkor a program mappájába kerül mentésre. Vizualizáció során nem szükséges folyamatosnak tűnő kép előállítása, ezért használható ez a módszer.

A program a színes háló előállításakor a forrásfájl attribútumait veszi alapul, majd figyeli, hogy egy elem hol szerepel, mik a kötődő elemei, illetve a gyerekei. Ez után egy listában szedi össze a node-ok színeit inputra és outputra. A színezés következtében a komplexitás lényegesen megnő, akár $O(n^2)$-re.

8. fejezet

Mintafeladat bemutatása

Az alábbiakban egy mintafeladat bemutatására kerül sor. A mintafeladat egy nagyon egyszerű BPEL példa. Egyszerűségét az indokolja, hogy a forrás file-nak és az elkészült hálónak is bele kell férni (olvashatóan) egy A4-es lap kereteibe. Továbbá, az egyszerűség segíti a megértést is.

8.1. A forrás file

Vegyük az alábbi forrást:

```
<?xml version="1.0" encoding="utf-8" ?>
<bpel:process name="hworld"
 targetNamespace="http://eclipse.org/bpel/sample"
 suppressJoinFailure="yes"
 xmlns:tns="http://eclipse.org/bpel/sample"
 xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
 >
  <bpel:import location="hworldArtifacts.wsdl"
        namespace="http://eclipse.org/bpel/sample"
        importType="http://schemas.xmlsoap.org/wsdl/" />

  <bpel:partnerLinks>

    <bpel:partnerLink name="client"
                partnerLinkType="tns:hworld"
                myRole="hworldProvider"
                partnerRole="hworldRequester"
                                                        />
  </bpel:partnerLinks>
  <bpel:variables>
    <bpel:variable name="input"
              messageType="tns:hworldRequestMessage"/>
    <bpel:variable name="output"
              messageType="tns:hworldResponseMessage"/>
  </bpel:variables>
```

```
  <bpel:sequence name="main">
    <bpel:flow name="mainFlow">
      <bpel:invoke name="flow1" operation="main1"/>
      <bpel:invoke name="flow2" operation="main2"/>
    </bpel:flow>

  </bpel:sequence>
  <bpel:sequence name="main1">
    <bpel:receive name="receiveInput" partnerLink="client"
           operation="initiate" variable="message"
           createInstance="yes"/>
  </bpel:sequence>

  <bpel:sequence name="main2">
    <bpel:send name="sendOutput" partnerLink="client"
               operation="initiate" variable="message"/>
  </bpel:sequence>
</bpel:process>
```

Ha megvizsgáljuk az inputot látjuk, hogy három szegmensből áll:

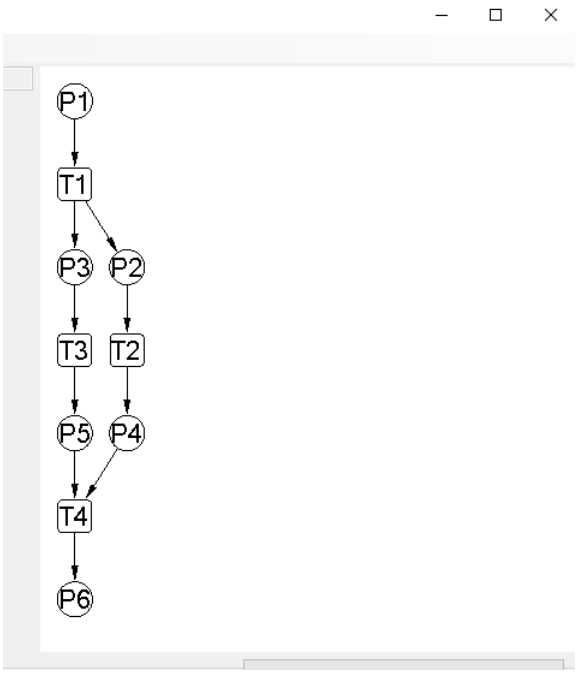
- *Meta:* Ide kerülnek a file definíciós metaadatok, ezek a BPEL engine-nek szólnak, a konverzióhoz (talán a név kivételével) szükségtelenek.
- *PartnerLinks* Partner linkek a BPEL folyamatok közötti összekötő eleme. Ha a konverzióhoz nem adunk meg adatokat a kapacitásokra, az elkészült hálóhoz, a partner link segítségével lehetne megtudni, milyen kapacitásokat kell az egyes node-okhoz rendelni.
- *Sequence: Main* Mint a C típusú nyelvekben itt is a *Main* a fő programszál. Alább felsorolható a többi segéd szál, illetve fölötte a változók, amiket a függvényeink, és különböző folyamatok használnak.

A bemenetünket megvizsgálva látjuk, hogy ez maximum három szint mély XML file (a Main-re nézve). Erre azért van szükség, mert a BPEL támogatja a kvázi végtelen mélységű file generálását, viszont a legtöbb XML feldolgozó rutin nem tud kezelni nagyon mély dokumentumokat. Ez azzal magyarázható, hogy az XML-re ilyenkor inkább, mint egy adatleíró, adattároló dokumentumra tekintenek, mintsem egy kvázi programkódra. A programomban ezért kellett egy megkötést alkalmazni. A három szintű mélység fixálása jelentősen megkönnyíti az input feldolgozását. Szerencsére a fixálás könnyen kivitelezhető, ugyanis a BPEL-ben is támogatott a saját metódus írása. Itt ún. szekvenciákkal (*sequence*) lehet ezeket leírni. A szekvenciákat pedig meg lehet hívni *invoke*-kal.
A folyamat maga lebontva:

- A main folyamat indul el először. A folyamat elindít egy két ágú párhuzamosítást a *flow*-al
- A *flow* két ágában egy-egy *invoke* meghívja a megfelelő szubrutint.
- A *main1* szubrutin majd rögtön elkezd várakozni egy üzenetre.
- A *main2* az előbb említett szubrutin számára küldi az üzenet tokent.

8.2. Az elkészült színes háló

Most, hogy láttuk a bemenetet, a programot "színes" módra kapcsolva nézzük meg az általa generált hálót. A program a következő hálót generálja: 8.1



8.1. ábra. A generált mintaháló

9. fejezet

Összefoglalás

A feladatot természetesen a BPEL megismerésével kezdtem. Majd miután egy alapvető képem kialakult a rendszerről, megpróbáltam saját magam is BPEL file létrehozását. Különböző próbálkozások után kialakult egy átfogó kép a folyamatokról, illetve a BPEL használhatóságáról. A következő lépés a Petri hálók megismerése volt. A programhoz szükség volt különböző modellekre, melyek a hálót írják le. A modelleket úgy kellett létrehozni, hogy rajtuk egyszerű legyen a tulajdonság vizsgálat, de ennek ellenére leírható legyen egy egész folyamat a teljesség igényével. A hálók vizsgálatához vissza kellett térni a diszkrét matematika, a matematikai logika és automaták és formális nyelvek tárgyból tanultakra.

Végeredményképp a program képes egy (pseudo) BPEL file feldolgozására. Az inputot konvertálja Petri hálóra a beállítások alapján, és az így kapott hálón elemzéseket hajt végre, és közli, hogy a háló véges-e, korlátos-e, illetve, van-e szükség belső puffer használatára. Az elkészült hálók a programból megtekinthetők, és rajtuk a tokenáram lépésről lépésre megvizsgálható. A hálómegjelenítés egyedüli korlátozója a képernyőméret, illetve a felbontás. A színes háló előállításakor, mivel nem egyértelmű a színezés ezért lehet többféle hálót is létrehozni. A program azonban csak egyet állít elő. Ezt azonban a definíciós szabálykészlet bővítésével könnyen lehet módosítani. Az önkényesen megválasztott színezés miatt egy, akár színes hálóra konvertálható folyamatot hamisan, konvertálhatatlanként kezel, de ez ha szükség adódna rá javítható több színezési algoritmus implementálásával. A fentebb említett gráfmegjelenítés szintén javítható egy olyan gráfrajzoló használatával, ami támogatja a vektoros formátumot, illetve a kapott kép átméretezését, vagy a részkép illesztéses megjelenítését.

Irodalomjegyzék

[1] Helloworld-bpeldesignerandode. `https://www.eclipse.org/bpel/users/pdf/HelloWorld-BPELDesignerAndODE.pdf`. Accessed: 2020.11.2.

[2] List of bpel engines. `https://en.wikipedia.org/wiki/List_of_BPEL_engines.htm`. Accessed: 2019-10-30.

[3] Visual studio editions. `https://visualstudio.microsoft.com/vs/compare/`. Accessed: 2021.05.16.

[4] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, et al. Business process execution language for web services, 2003.

[5] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of ws-bpel processes. In *International Conference on Service-Oriented Computing*, pages 269–282. Springer, 2005.

[6] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Towards distributed bpel orchestrations. *Electronic Communications of the EASST*, 3, 2007.

[7] Jorge Cardoso. Complexity analysis of bpel web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

[8] Gregory Hackmann, Mart Haitjema, Christopher Gill, and Gruia-Catalin Roman. Sliver: A bpel workflow process execution engine for mobile devices. In *International Conference on Service-Oriented Computing*, pages 503–508. Springer, 2006.

[9] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. volume 3649, pages 220–235, 09 2005.

[10] Oliver Kopp, Katharina Görlach, Dimka Karastoyanova, Frank Leymann, Michael Reiter, David Schumm, Mirko Sonntag, Steve Strauch, Tobias Unger, Matthias Wieland, et al. A classification of bpel extensions. *Journal of Systems Integration*, 2(4):3–28, 2011.

[11] Máté Kovács, Dániel Varró, and László Gönczy. Formal analysis of bpel workflows with compensation by model checking. *Computer Systems Science and Engineering*, 23(5):349–363, 2008.

[12] Jean B Lasserre and Philippe Mahey. Using linear programming in petri net analysis. *RAIRO-Operations Research*, 23(1):43–50, 1989.

[13] Philip Mayer and Daniel Lübke. Towards a bpel unit testing framework. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 33–42. ACM, 2006.

[14] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[15] Chun Ouyang, Marlon Dumas, Stephan Breutel, and Arthur ter Hofstede. Translating standard process models to bpel. In *International Conference on Advanced Information Systems Engineering*, pages 417–432. Springer, 2006.

[16] Chun Ouyang, Eric Verbeek, Wil MP van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur HM ter Hofstede. Wofbpel: A tool for automated analysis of bpel processes. In *International Conference on Service-Oriented Computing*, pages 484–489. Springer, 2005.

[17] Florian Rosenberg and Schahram Dustdar. Business rules integration in bpel-a service-oriented approach. In *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, pages 476–479. IEEE, 2005.

[18] Aleksander Slominski. Adapting bpel to scientific workflows. In *Workflows for e-Science*, pages 208–226. Springer, 2007.

[19] Wil MP Van Der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *Information and Software Technology*, 50(3):131–159, 2008.

Melléklet használati útmutatója

A dolgozat PDF változata (dolgozat.pdf), és az előállításához szükséges LaTeXforrásfájlok megtalálhatóak a mellékleten.

A mellékelt programok futtatásához Windows operációs rendszer és a .NET keretrendszer 4.8 as verziója szükséges. A Python kódok Python 3.6 alatt lettek tesztelve.

A program az .exe állománnyal indítható el. A program tartalmaz help provider-t, azaz futás közben a segítség használatával a program használata megismerhető.