Szakdolgozat

Miskolci Egyetem

Rutinszerű feladatok automatizálása grafikus felhasználói felületek esetében

Készítette:

Pázmándi Erik

Programtervező informatikus

Témavezető:

Dr. Kovács Béla

Konzulens:

Piller Imre

Miskolc, 2022

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

Szakdolgozat Feladat

Pázmándi Erik (GXN833) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Folyamatelemzés, RPA

A szakdolgozat címe: Rutinszerű feladatok automatizálása grafikus felhasználói felületek esetében

A feladat részletezése:

A számítógépek kifejlesztésének és használatának egyik fő motivációja, hogy a segítségével az automatizált módon végrehajtható folyamatok emberi beavatkozás nélkül is végrehajthatóak legyenek. Ennek ellenére számos esetben tapasztalhatjuk, hogy az alkalmazások felhasználói felületén rutinszerűen, repetitíven hajtanak végre műveleteket.

A dolgozat azt vizsgálja, hogy ezek a folyamatok a korábban rögzített eseménysorok alapján hogyan ismerhetők fel. Bemutatja az RPA (Robotic Process Automation) eszközkészletét, többek között a folyamatelemzés elterjedt módszereit, alkalmazási lehetőségeit, a grafikus felhasználói felületekhez kapcsolódó speciális eseteket. Az elemzésekhez, automatizálást segítő eszköz elkészítéséhez Microsoft Windows platformon Delphi programozási nyelv kerül felhasználásra.

Témavezető: Dr. Kovács Béla (egyetemi docens)

Konzulens: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje: 2021. Szeptember 23.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Pázmándi Erik**; Neptun-kód: `GXN833` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Rutinszerű feladatok automatizálása grafikus felhasználói felületek esetében* 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 dolgozat azt vizsgálja, hogy a számítógépek felhasználói felületén miféle sokszor elismételt folyamatok zajlanak le, ezek egy robot szempontjából hogyan is néznek ki, hogyan lehet ezeket utánozni, illetve automatizálni. Bemutat folyamatelemzési módszereket, ... BŐVÍTENI! (1-2 oldal)

2. fejezet

Koncepció

2.1. Alpha-algoritmus

Az Alpha algoritmus (vagy Alpha bányász) mint folyamatelemzési algoritmus célja, hogy eseménysorozatok halmazából egy ok-okozat rendszert építsen fel. Először van der Aalst, Weijters és Măruşter hozta be a köztudatba. A működésében az eseménysorok halmazát nevezhetjük eseménynaplónak is. Ez az eseménynapló úgynevezett trace halmazoknak a halmaza, egy trace pedig adott tevékenységnek a sorozata.

Az Alpha bányász volt a legelső folyamatbányászati módszer amit valaha javasoltak és egy egész jó rálátást biztosít a folyamatbányászat céljára, valamint arra, hogy a folyamatokban lévő különböző tevékenységek hogyan is vannak végrehajtva. Emelett, az Alpha bányász szolgált számos újabb folyamatbányászati technika (pl.: Heurisztikus bányász, genetikus bányászat) alapjaként.

2.1. definíció. *(Munkafolyamati trace)* Egy string a T ábécé feladatai közül.

2.2. definíció. *(Munkafolyamati napló)* Munkafolyamati tracek halmaza.

2.1.1. Rövid leírása

Az algoritmus egy munkafolyamati naplót $W \subseteq T^*$ kap bemenetként, és eredményként egy munkafolyamati hálót épít fel.

Ezt az alapján csinálja meg, hogy megvizsgálja az általános kapcsolatokat az egyes feladatok között. Például egy adott feladat lehet, hogy minden esetben megelőz egy másik feladatot, ami egy hasznos információ.

2.1.2. Eseménynapló

Az eseménynapló az elsődleges szükséglet bármely folyamatbányászati algoritmus alkalmazásához. Az eseménynapló a következőket tartalmazza: egyedi azonosító az esethez, tevékenység megnevezése valamint egy időbélyeg. Egy eseménynaplót akár tevékenységek halmazának halmazaként is lehet ábrázolni.

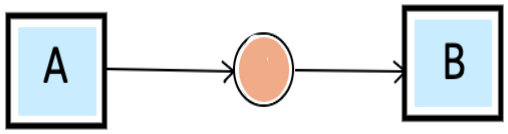
Az Alpha bányász szabályai szerint az egyes tevékenységek között az alábbi 4 féle kapcsolat egyike lehetséges:

1. **Közvetlen sorrend:** $x > y$ akkor és csakis akkor ha az x eseményt közvetlenül követi y.

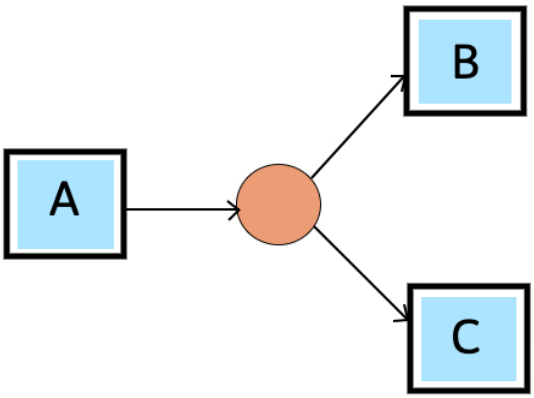
2. **Okozat:** $x \to y$ ha $x > y$ és nem $y > x$.
3. **Párhuzam:** $x \parallel y$ ha $x > y$ és $y > x$.
4. **Választás:** $x \# y$ ha nem $(x > y)$ és nem $(y > x)$.

2.1.3. Minták

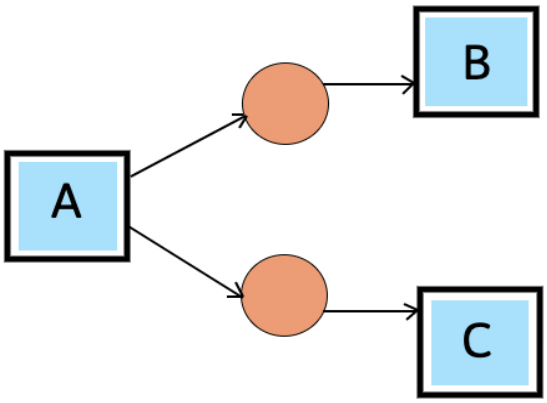
2.1. ábra. **Szekvencia: A $\to$ B**



2.2. ábra. **XOR-elágazás: A $\to$ B, A $\to$ C és B $\#$ C**



2.3. ábra. **ÉS-elágazás: A → B, A → C és B ‖ C**



2.1.4. Példa

Vegyük példának a következő eseménynaplót:

2.4. ábra. Példa eseménynaplpó

ID	Tevékenység	Időbélyeg
1	A	2022-10-05 13:50:40.000
1	B	2022-10-05 16:30:12.000
1	C	2022-10-05 16:57:31.000
1	D	2022-10-06 13:50:41.000
2	A	2022-10-06 15:30:27.000
2	C	2022-10-06 16:23:33.000
2	B	2022-10-07 08:33:02.000
2	D	2022-10-07 12:41:11.000
3	A	2022-10-07 13:02:57.000
3	E	2022-10-07 14:11:21.000
3	D	2022-10-07 14:59:22.000

Ebben az esetben az eseménynaplót az alábbi módon tudjuk jelölni:

$$L_1 = [< A, B, C, D >, < A, C, B, D >, < A, E, D >]$$

Az Alpha bányász úgy kezdi a munkát, hogy az eseménynaplót közvetlen-sorrend, okozat, párhuzam és választás relációkra alakítja és ezeket felhasználva létrehoz egy petri hálót ami leírja a folyamat modellét.

Első lépésként létrehoz egy lenyomati mátrixot:

2.5. ábra. Példa lenyomati mátrix

	A	B	C	D	E
A	#	→	→	#	→
B	←	#	‖	→	#
C	←	‖	#	→	#
D	#	←	←	#	←
E	←	#	#	→	#

Y_W az összes (A, B) pár halmaza a feladatok maximális halmazából úgy, hogy:

- Egyik $A \times A$ és $B \times B$ sem tagja $>$-nek, és
- $A \times B$ részhalmaza $\rightarrow$-nek.

P_W tartalmazza az egyes Y_W-hez tartozó helyeket $p_{(A,B)}$, plussz a beviteli i_W helyet és a kimeneti o_W helyet. A folyamati reláció F_W az alábbiak uniójából áll össze:

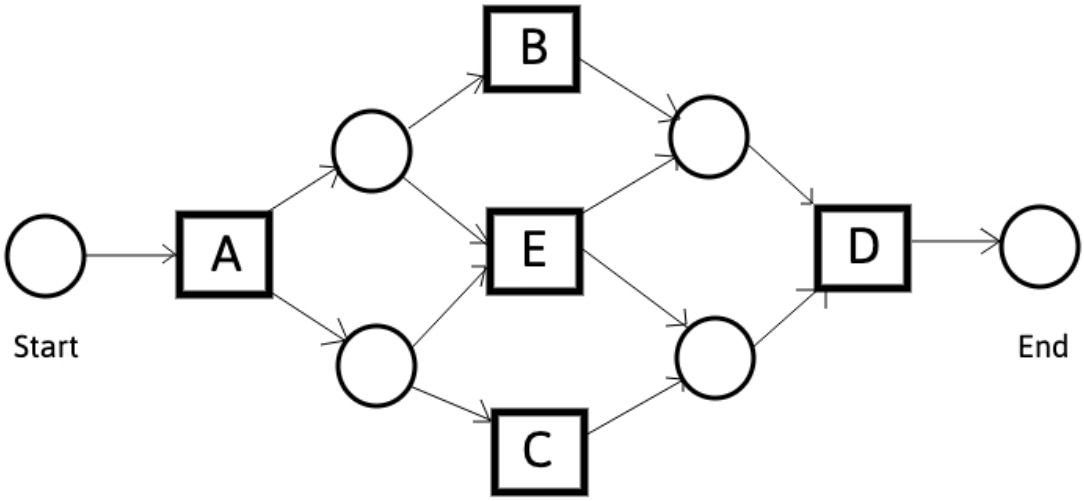
- $\{(a, p_{(C,B)})|(A, B) \in Y_W \wedge a \in A\}$
- $\{(p_{(A,B)}, b)|(A, B) \in Y_W \wedge b \in B\}$
- $\{(i_W, t)|t \in T_1\}$
- $\{(t, i_0)|t \in T_0\}$

Az eredmény

- egy Petri háló struktúra $\alpha(W) = (P_W, T_W, F_W)$
- egy beviteli hellyel i_W és egy kimeneti hellyel o_W
- mivel minden T_W átmenet F_W-úton van i_W-ből o_W-be, így valóban egy munkafolyamati háló.

Ehhez a példához az alábbi petri háló jön létre az Alpha bányász használatával

2.6. ábra. Példa kimeneti petri háló



2.1.5. Korlátozások

- **Implicit helyek**: Az Alpha bányász nem tud különbséget tenni az implicit és a szükséges helyek között, így a felfedezett petri hálóban előfordulhatnak plusz szükségetelen helyek.
- **Ciklusok**: Az Alpha bányász nem képes 1-gyes és 2-tes hosszúságú ciklusok felismerésére a folyamatmodellben.
- A helyi függőségeket gyakran nem veszi észre az Alpha bányász.

Forrás: (Alpha-algoritmus, 2022)

2.2. Robotic Process Automation

A Robotikus Folyamatautomatizálás (továbbiakban: RPA) egy olyan szoftvertechnológia, mely lehetővé teszi, hogy az erre specializált szoftverek emberi felhasználót emulálva lépjenek kapcsolatba a számítógépek digitális felületeivel.

Minden egyes ilyen szoftvernek más az eszköztára, van amelyik azt tudja értelmezni, hogy mi van a képernyőn, van amelyik felismer és kinyer adatokat, viszont abban mindegyik osztozik, hogy adott lépésekből meghatározott folyamatokat hajt végre.

Összefoglalva, egy ilyen tökéletesített rendszer ugyanazt tudja mint egy felhasználó, viszont sokkal gyorsabban és konzisztensebben, anélkül, hogy fel kellene állnia nyújtózni vagy elmenni egy kávészünetre.

2.2.1. Alkalmazási területek

Lényegében bármely olyan modern cég tudja hasznosítani ezt a technológiát, mely számítógépet használva pl. nyilvántartást vezet, pénzügyeit digitálisan kezeli, alapvetően a digitális térben mozog, stb..., tehát bárhol ahol embereket digitális tevékenységül alul fel lehet szabadítani.

Elsősorban az adott cégtől függ, hogy belevág-e egy ilyen szoftvertechnológiás megoldásba, de íme néhány szektor ahol alkalmazható, vagy már alkalmazásra is került:

- Egészségügy
- Telekommunikáció
- Gyártástechnológia
- Állami szektor
- Kereskedelem
- Pénzügyi szolgáltatások

Gyakorlatilag csupán az adott folyamattól függ, hogy lebontható-e olyan triviális lépésekre, melyeket már az RPA eszközkészletével automatizálni lehet. Természetesen ahogy fejlődik ez a technológia, úgy egyre nagyobb százalékban lehet majd ezeket is automatizáltnak tekinteni.

Alább található néhány mai rendszer, melyeket a technológia úttörőjének lehet nevezni:

1. UIPath
2. Microsoft Power Automate
3. Blue Prism
4. Automaton Anywhere
5. Kofax

2.2.2. A technológia jövője

Mivel egy automatizációs technológia jövőjéről van szó, ezért az RPA nem egy kis dolog ami feltehetően el fog tűnni, hanem nagy valószínűséggel befolyásolni fogja a munkaerőpiac jövőjének egészét..... (folytatni / elvetni?)

2.3. Delphi

A dolgozathoz készült szoftver Delphi nyelven íródott, így fontos legalább nagyvonalakban ismerni a nyelvet, hogy tudjuk miért is.

A Delphi egy általános célú erősen típusos objektum orientált programozási nyelv és szoftvertermék ami az Object Pascal programozási nyelv Delphi dialektusát használja, integrált fejlesztői környezetet biztosít, újabban a gyors alkalmazásfejlesztés (RAD, Rapid application development) szoftverfejlesztési elv szerint. A Delphi compilerei natív

kódot generálnak a célrendszertől függően, legyen az Microsoft Windows, macOS, iOS, Android vagy Linux (x64).

(Embarcadero, 2022)

2.3.1. Múltja röviden

Az „anyanyelve" a Delphinek a Pascal, ami pedig a modellje nagy részét az Algolnak köszönheti - az első magas-szintű progamozási nyelvnek ami olvasható, struktúrált és szisztematikusan meghatározott szintaxissal rendelkezik. A hatvanas években számos utódját fejlesztették az Algolnak, ezek közül a legsikeresebb a Pascal volt.

1983-ban jelent meg az első Turbo Pascal a Borland jóvoltából, ami már integrált fejlesztői környezettel rendelkezett. 1995-ben vezették be a RAD szoftverfejlesztési elvre épülülő környezetet, amit Delphi-nek neveztek, ezáltal átalakítva a Pascal nyelvet egy objektum-orientált vizuális programozási nyelvvé. A célja ennek elsősorban az volt, hogy ennek az új terméknek központi részét képezzék az adatbázis eszközök és kapcsolatok.

2006-ban a Borland átadta a fejlesztőeszközöket a CodeGear nevű leányvállalatának, majd ezt a leáanyvállalatot 2008-ban eladta az Embarcadero Technologies-nek. Ez az új cég megtartotta a régi fejlesztői divíziót és számos új verziót dobott piacra. 2015-ben az Idera Software nevű cég pedig felvásárolta az Embarcadero-t és mind a mai napig ugyanúgy Embaracadero márka alatt működteti a fejlesztői eszközök divízióját.

Az évek alatt rendkívül sok modernizáción ment át a Delphi. OLE automatizáció és változó adattípus támogatásától kezdve, DLL debugoláson és XML támogatáson keresztül egészen a multi-platform alkalmazásokig és az in-line változó deklarálásig.

(Zarko, 2021)

2.3.2. Napjainkban

Sajnos számos rosszul időzített és rosszul kivitelezett marketing döntés miatt a 2000-es évektől kezdve a Delphi kifejezetten kiesett rengeteg programozó kedvelt programozási nyelve közül, azonban az elmúlt néhány évben ismét sikerült egyre nagyobb ismerettségre szert tennie a komolyabb fejlesztők körében.

Bár közel sem a legelterjedtebb nyelv, számos előnnyel rendelkezik sok másikkal szemben. Ilyenek például az alábbiak:

1. **Könnyen olvasható kód**: Már eredetileg a Pascal megalkotásánál az egyik fő cél az volt, hogy oktatási célra lehessen használni, emberi szemmel is könnyen olvasható legyen a komplex alacsony-szintű kód. Erre egy nagyon jó példa a "{" és "}" karakterek (amiket csak a memóriával való spórolás miatt jelöltek így) "begin" és "end" kulcsszóra való cseréje.

2. **Multi-platformitás**: A megfelelően megírt (OS-független) kódot néhány kattintással le lehet fordítani a legismertebb operációs rendszerek natív kódjára.

3. **Natív kód**: Az alkalmazás lefordításával natív kódot kapunk, ami azért előny, mert semmilyen egyéb keretrendszer telepítésére nincsen szükségem (pl.: MS C++ Runtime Environment, Java Runtime Environment, stb...)

4. **Adatbázis támogatás**: Számos adatbázis kapcsolatot és adatfeldolgozási módszert beépített módon támogat.

5. **Fordítási sebesség**: A mai napig az egyik leggyorsabb a fordítási sebessége más fejlesztőeszközökhöz képest, ezáltal felgyorsíva magát a fejlesztési és debugolási folyamatokat.

2.3.3. Delphi a dolgozathoz

Az előző alfejezetben felsoroltaknak megfelelően kiderült, hogy a Delphi az egy kifejezetten robosztus és erőteljes programozási nyelv. Ez sok nyelvről elmondható természetesen, viszont az alábbi két pont miatt került kiválasztásra a dolgozathoz:

- **Modern Windows API**: A fejlesztett szoftver (mint a legtöbb RPA eszköz) közvetlen viszonyban van a Windows API-val, hiszen az operációs rendszer teszi elérhetővé az egyes erőforrásokat (pl. rögzíteni a felhasználó bevitelét még akkor is ha a program háttérben van), illetve teszi lehetővé az input injektálását a folyamat visszajátszásához.
- **Natív kód**: Mivel natív kódra kerül fordításra a programkód, ezért bármely Windows (Vista vagy újabb) operációs rendszerrel rendelkező számítógépen futtatható a program bármiféle keretrendszer nélkül.

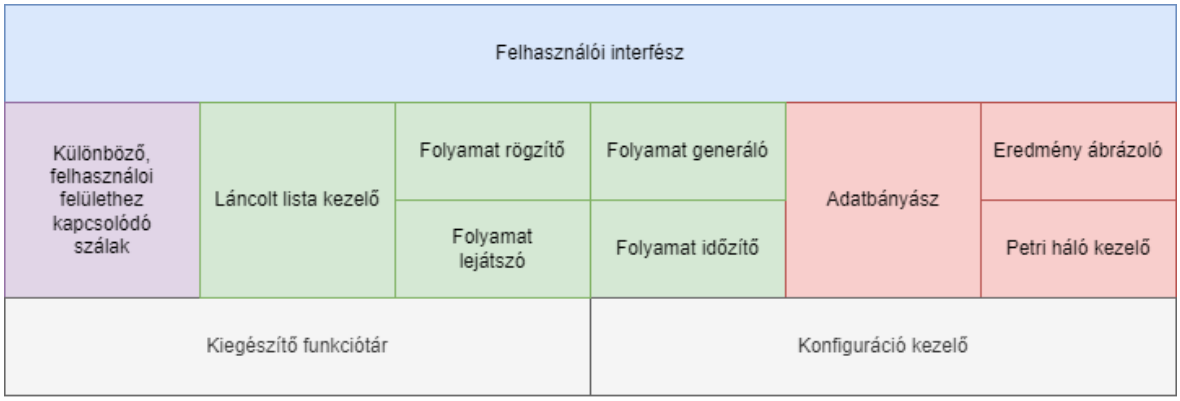
3. fejezet

Tervezés

A dolgozat által vizsgált témahoz egy komplex multifunkciós szoftver került megtervezésre, mely a dolgozati téma elemzési részében kifejezetten nagy szerepet tölt be. Összetettsége révén rengeteg időt és odafigyelést igényelt már maga a tervezési fázis is. Számos ábra és tervezet került megalkotásra, melynek a túlnyomó része rendkívül jelentősnek bizonyult az implementáció során.

A legmagasabb szinten az alábbi ábra nyújtja a legtisztább áttekintését a különböző funkcióknak és a szoftver sokszínűségének.

3.1. ábra. **High-level áttekintő ábra**



- **Felhasználói interfész**: A kezelőfelület, amivel a felhasználó eléri és kezelni tudja az egyes szoftverfunkciókat.
- **Felhasználói felülethez kapcsolódó szálak**: Fontos - a felhasználó számára nem látható - szálak, amelyek feldata bizonyos billentyűkombinációk figyelése anélkül, hogy a program reszponzivitását kártékonyan befolyásolnák.
- **Láncolt lista kezelő**: Az egyes folyamatok láncolt listaként vannak kezelve a szoftverbe, ez az alrendszer felel a megfelelő értelmezésükért.
- **Folyamat rögzítő**: Figyeli és rögzíti a perifériák általi beviteli értékeket.
- **Folymat generáló**: Előre meghatározott forgatókönyvek alapján úgy generál folyamatokat mintha azt egy felhasználó végezte volna el.

- **Folyamat lejátszó**: A folyamatokat játsza vissza, egy felhasználót szimulál.
- **Folyamat időzítő**: A Windowsba integrált rendszert felhasználva ütemez / időzít folyamatokat.
- **Adatbányász**: Az Alpha-algoritmust implementálva folyamatelemzést hajt végre több folyamaton.
- **Eredmény ábrázoló**: Az Adatbányász által elvégzett folyamatelemzés eredményeit jeleníti meg.
- **Petri háló kezelő**: A petri-hálót mint struktúra, valamint a hozzátartozó függvényeket a szoftver számára értelmezhető módon implementálja.
- **Kiegészítő funkciótár**: Számos hasznos funkció gyűjteménye, melyet a többi alrendsze használ.
- **Konfiguráció kezelő**: Futásidők között felhasználói preferenciák és beállítások tárolásáért és betöltéséért felel.

3.1. Az Alpha-algoritmus

Az Alpha-algoritmust mint folyamatelemzési módszert hasznosan lehet alkalmazni a dolgozati témában. Az algoritmus, jelentőségét tekintve elengedhetetlen részét képezi a dolgozatnak. Jelen esetben a folyamatokat azok jellegétől és céljától függetlenül lehet elemezni, akár cél nélküli beviteli sorozatra is alkalmazható az Alpha-algoritmus.

3.1.1. Alkalmazása

Ebben az alfejezetben bemutatásra kerül, hogy hogyan is kapcsolódik pontosan az Alpha-algoritmus a dolgozat témájához.

Először is pontosan meg kell határozni, hogy milyen lépésekből áll az algoritmus.

3.1. definíció. *(α-algoritmus):* Legyen L egy eseménynapló adott E események halmaza felett. Ekkor a kimeneti $\alpha(L)$ petri hálót az alábbi módon határozzuk meg:

1. Definiáljuk az összes eseményt.

$$E_L = \{e \in E | \exists_{\sigma \in L} e \in \sigma\},$$

2. Definiáljuk az összes bemeneti eseményt.

$$E_I = \{e \in E | \exists_{\sigma \in L} e = first(\sigma)\},$$

3. Definiáljuk az összes kimeneti eseményt.

$$E_O = \{e \in E | \exists_{\sigma \in L} e = last(\sigma)\},$$

4. Kiszámítjuk az összes lehetséges A és B halmazt úgy, hogy az összes esemény A-ban és B-ben függetlenek legyenek egymástól, valamint minden A-beli esemény okozati kapcsolatban álljon B-beli eseményekhez.

$$\begin{aligned} X_L = &\{(A, B) | A \subseteq E_L \wedge A \neq \emptyset \wedge B \subseteq E_L \wedge B \neq \emptyset \wedge \\ &\forall_{a \in A} \forall_{b \in B} a \rightarrow_L b \wedge \forall_{a1,a2 \in A} a_1 \#_L a_2 \wedge \forall_{b1,b2 \in B} b_1 \#_L b_2\}, \end{aligned}$$

5. Elhagyjuk a nem-maxmiális halmazokat.

$$Y_L = \{(A, B) \in X_L | \forall_{A',B' \in X_L} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A, B) = (A', B')\},$$

6. Helyeket rendelünk az összes származtatott halmazhoz valamint a kezdő- és végállapotokhoz.

$$P_L = \{p_{A,B} | (A, B) \in Y_L\} \cup \{i_L, o_L\},$$

7. Berajzoljuk a kapcsolatokat.

$$F_L = \{(a, p_{(A,B)}) | (A, B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) | (A, B) \in Y_L \wedge b \in B\} \cup \{(i_L, e) | e \in E_I\} \cup \{(e, o_L) | e \in E_O\},$$

8. Visszatérünk a petri hálóval.

$$\alpha(L) = (P_L, E_L, F_L).$$

Forrás: (Aalst & Dongen, 2013)

3.2. példa. Ebben a példában három előre létrehozott folyamaton kerül alkalmazásra az Alpha-algoritmus. A folyamatok egyszerűek, hogy szemléletes legyen a példa, viszont ugyanezzel a módszerrel több száz vagy akár több ezer hosszú folyamaton is alkalmazható az algoritmus.

Maguk a folyamatok szolgálnak bemenetként, részeredményekként eseménynapló és lenyomati mátrix jön létre, kimentként pedig egy olyan petri háló kerül generálásra mely leírja a folyamat modelljét.

3.2. ábra. Beviteli folyamatok

Folyamat	ID	Típus	Érték	Érték típusa	Eltelt idő
1	1	Key	Left Alt	WM_SYSKEYDOWN	0
1	2	Key	F4	WM_SYSKEYDOWN	100
1	3	Key	F4	WM_SYSKEYUP	150
1	4	Key	Left Alt	WM_KEYUP	612
2	1	Key	Left Alt	WM_SYSKEYDOWN	0
2	2	Key	F4	WM_SYSKEYUP	80
2	3	Key	F4	WM_SYSKEYDOWN	51
2	4	Key	Left Alt	WM_KEYUP	152
3	1	Key	Left Alt	WM_SYSKEYDOWN	0
3	2	Mouse	25:1022	WM_LBUTTONDOWN	151
3	3	Key	Left Alt	WM_KEYUP	188

Az Alpha-algoritmus alkalmazásában, mint bármely folyamatbányászati algoritmusnál, első lépésként ezekből az eseményekből fel kell építeni az eseménynaplót amiből

később dolgozik az algoritmus. Ez a lépés konkrétan arról szól, hogy a már meglévő folyamatok az Alpha-algoritmusnak szükséges formátumra kerülnek átalakításra.

Ez jelen esetben az alábbi három szabály alapján történik:

1. A "**Folyamat**" elnevezésű oszlop alapján triviális módon meghatározásra kerül az esethez tartozó egyedi azonosító,
2. A "**Típus**", "**Érték**" és "**Érték típusa**" oszlophármas értékeiből létrejön a tevékenység megnevezése, ami a továbbiakban „T_n"-ként lesz feltüntet-ve,
3. Az "**Eltelt idő**" oszlop alapján (az előző esemény óta eltelt időt mutatja) pedig létrejön egy relatív-időbélyeg az "**ID**" oszlop segítségével, hiszen az utóbbi alapján határozható meg az események szekvenciája.

Ezeknek megfelelően az alábbi eseménynaplót kapjuk:

3.3. ábra. Eseménynapló

Azonosító	Tevékenység	Relatív időbélyeg
1	T_0	0
1	T_1	100
1	T_2	250
1	T_3	762
2	T_0	0
2	T_2	80
2	T_1	131
2	T_3	283
3	T_0	0
3	T_4	151
3	T_3	339

Miután megvan az eseménynapló, a következő két lépésben meghatározzuk a bemeneti- és kimeneti események halmazait:

1. $E_I =< T_0 >$
2. $E_O =< T_3 >$

Ezután a következő lépéshez az eseménynaplóban szereplő események kapcsolatait közvetlen-sorrend, okozat, párhuzam és választás relációkra alakítja az algoritmus.

Ezzel jön létre az alábbi lenyomati mátrix:

3.4. ábra. Lenyomati mátrix

	T_0	T_1	T_2	T_3	T_4
T_0	#	$\rightarrow$	$\rightarrow$	#	$\rightarrow$
T_1	$\leftarrow$	#	$\|\|$	$\rightarrow$	#
T_2	$\leftarrow$	$\|\|$	#	$\rightarrow$	#
T_3	#	$\leftarrow$	$\leftarrow$	#	$\leftarrow$
T_4	$\leftarrow$	#	#	$\rightarrow$	#

Ezen a mátrixon kerül ábrázolásra az összes esemény közötti kapcsolat. Több szempontból is hasznos ez a mátrix, többek között a struktúrája is megfelelő ahhoz, hogy program szinten meghatározzuk a következő lépésben a lehetséges halmazpárokat, valamint emberi szemmel is kifejezetten könnyen értelmezhető.

Ezt a mátrixot felhasználva az alábbi halmazpárok lehetségesek a jelenlegi példában:

3.5. ábra. Lehetséges halmazpárok

A	B
$\{T_0\}$	$\{T_1\}$
$\{T_0\}$	$\{T_2\}$
$\{T_0\}$	$\{T_4\}$
$\{T_1\}$	$\{T_3\}$
$\{T_2\}$	$\{T_3\}$
$\{T_4\}$	$\{T_3\}$
$\{T_0\}$	$\{T_1, T_4\}$
$\{T_0\}$	$\{T_2, T_4\}$
$\{T_1, T_4\}$	$\{T_3\}$
$\{T_2, T_4\}$	$\{T_3\}$

Következő lépésként ezekből a halmazpárokból kell eltávolítani a nem-maximálisakat, azaz azokat amik részhalmazai egy másiknak. Ezt program szinten egy többszörös ciklus segítségével könnyedén el lehet végezni, jelen példában pedig az alábbi négy halmazpár maradt:

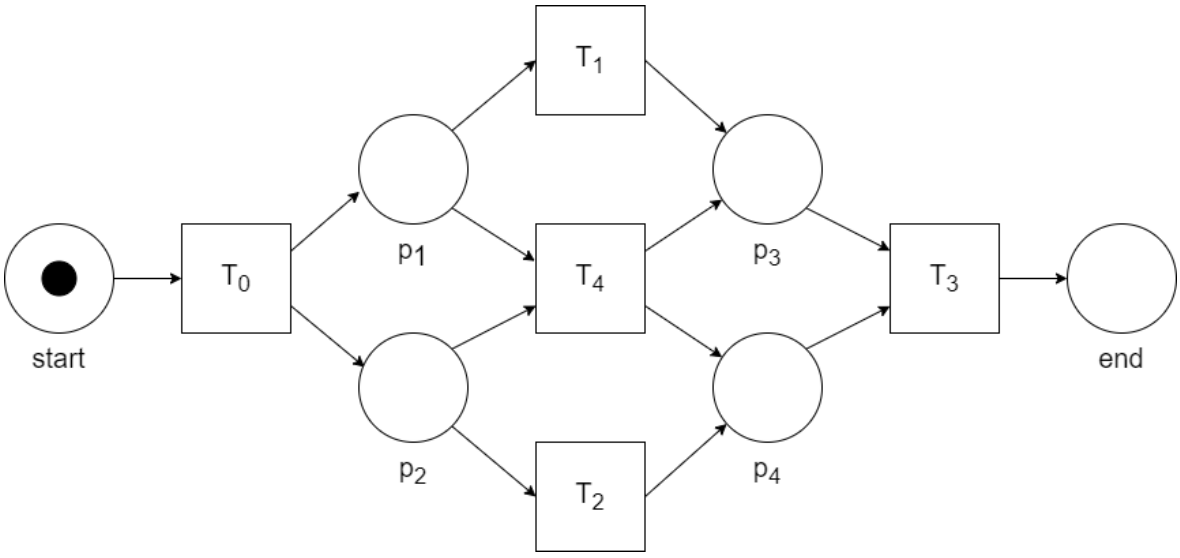
3.6. ábra. Maradék halmazpárok

A	B
$\{T_0\}$	$\{T_1, T_4\}$
$\{T_0\}$	$\{T_2, T_4\}$
$\{T_1, T_4\}$	$\{T_3\}$
$\{T_2, T_4\}$	$\{T_3\}$

Miután ezek a halmazpárok meghatározásra kerültek, helyek ($p_1 - p_4$) lesznek hozzájuk rendelve. Ezekhez a helyekhez létrehozásra kerülnek a megfelelő bemeneti és kimeneti átmenetek, valamint a végső bemeneti és kimeneti állapotok is.

Amint ez megvan, berajzolásra kerülnek a kapcsolatok is, a végén pedig a következő petri háló kerül megjelenítésre.

3.7. ábra. **Kimeneti petri háló**



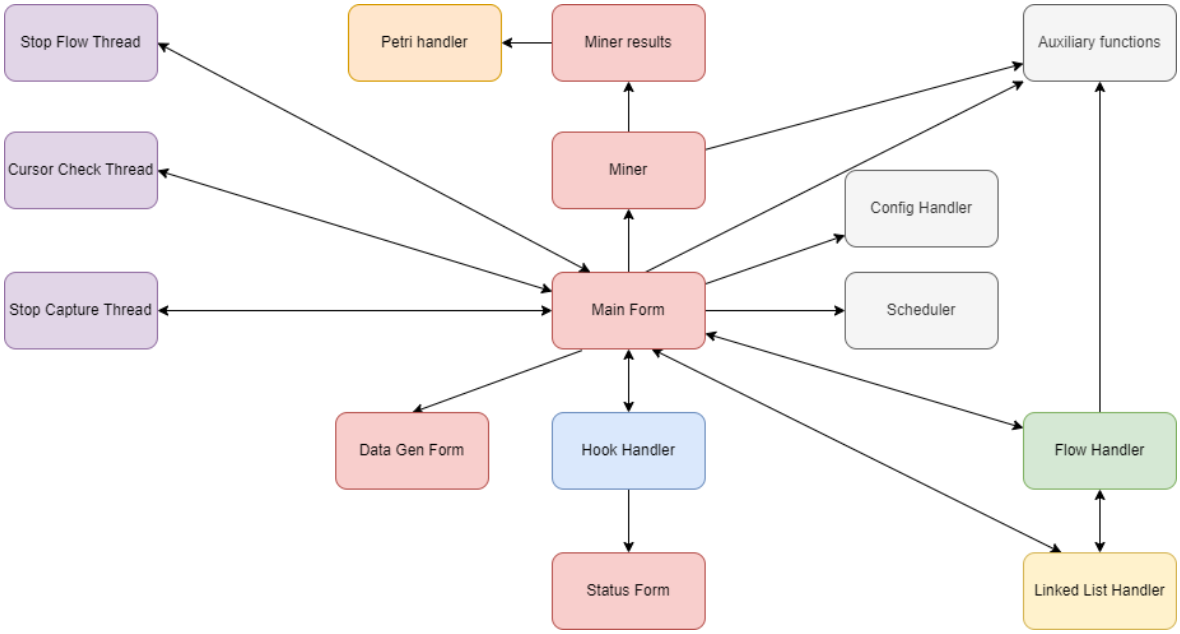
4. fejezet

Megvalósítás

A program gyakorlati megvalósítása egy-két apróságtól eltekintve megegyezik a tervezettel. Az alapvető stuktúra az tartva lett, az egymáshoz tartozó funkciók, változók és kódrészletek külön egységekbe (továbbiakban: unitok) lettek szedve, így csoportosítva az egyes szoftverfunkciókat.

Ezeknek a unitok többféle kapcsolatban állhatnak egymással, az egymásra való hivatkozásuknak függvényében. Ezeket a kapcsolatokat a következő ábra demonstrálja.

4.1. ábra. Unitok kapcsolatai



Pirossal azok a unitok vannak ábrázolva melyekhez tartozik grafikus felület is, a további színek pedig az egyes funkciók csoportosítására szolgálnak. Részletesebben a unitokról:

- **Unit_Main**: Ez a program főegysége, ebben került implementálásra a főablak és annak felhasználói felülete, számos funkciót lát el:

  1. Implementálja a felhasználói felület objektumait, azoknak a tulajdonságait és rutinjait,

2. Inicializálja a programot futtatásnál, betölti a felhasználó konfigurációit,
3. Kezeli a program rendszertálcára való kicsinyítését,
4. Kezeli azt a néhány globális változót amik szükségesek (pl. a futtatható fájl elérési útvonala).

- **Unit_ConfigHandler**: Konfigurációkezelő osztályt és az abba tartozó objektumokat és rutinokat implementálja, melyek lehetővé teszik a futásidő alatti konfigurációs beállítások titkosítva történő tárolását.

```
TConfigHandler = class(TObject)
  configFile: string;
  configArray : array of array of string;

public
  procedure Save(Key: string; Value: string);
  function Load(Key: string; Fallback: string): string;

  constructor Create(filePath: string);
  destructor Destroy; override;

  function Encrypt(const value: string): UnicodeString;
  function Decrypt(const value: string): UnicodeString;

  function MatchCharToArrayIndex(const character: char):
     integer;
end;
```

- **Unit_AuxiliaryFunctions**: Néhány olyan kiegészítő rutint tartalmazó gyűjtemény, melyeket a többi egység használ. Külön egységbe ki lettek gyűjtve, hogy az OOP alapelvek teljesüljenek.

- **Unit_StopFlowThread**: Egy olyan szálat implementáló egység, amely feladata kifejezetten a **F2** + **F3** billentyűkombináció lenyomásának felügyelete.

  Ez a billentyűkombináció felel azért, hogy az éppen futó folyamatot le lehessen futás közben állítani. Feltétlenül szükséges, hogy ez ilyen módon legyen implementálva, hiszen ennek a billentyűkombinációnak akkor is működnie kell, ha a fókusz éppen egy másik alkalmazáson van.

  A Windows API és a **Unit_Main** által implementált rutinokra és változókra hivatkozik a működése során.

```
procedure TStopFlowThread.Execute;
begin
  repeat
    if (GetKeyState(VK_F2) < 0) and (GetKeyState(VK_F3) < 0) then
        begin
      Form1.Btn_StartFlowClick(stopFlowThread);
    end;
  until not runStopFlow;
end;
```

- **Unit_CursorCheckThread**: Egy olyan szálat implementáló egység, melynek feladata a kurzor aktuális pozíciójának a felhasználói felületen történő frissítése. Windows API-t meghívva jut hozzá a szükséges adathoz.

```
procedure TCursorCheckThread.Execute;
var
  p: TPoint;
begin
  FreeOnTerminate := true;
  repeat
    GetCursorPos(p);
    Form1.Lab_Cursor_X.Caption := 'x: ' + IntToStr(p.X);
    Form1.Lab_Cursor_Y.Caption := 'y: ' + IntToStr(p.Y);
  until not runCursorPos;
end;
```

Ennek a célja, az, hogy amikor a felhasználó kézileg akar egérkattintást hozzáadni az aktuális folyamathoz, akkor megkönyítse a megfelelő képernyő-koordináták meghatározását.

- **Unit_StopCaptureThread**: Egy olyan szálat implementáló egység, melynek feladata hasonló az előzőekben bemutatott **TStopFlowThread**-éhez.

  Az eltérés annyi, hogy az **F2** + **F4** billentyűkombinációt figyeli, melynek lenyomására egy olyan rutint hív meg, mely leállítja az éppen futó felhasználói eseménysor rögzítését.

- **Unit_LinkedListHandler**: Egy olyan struktúrát implementál, melyre a szoftver összes többi folyamatkezezlő egysége épül. Az ebből a láncolt lista struktúrából példányosított elemek tartalmazzák a folyamati lépésekhez tartozó összes információt.

  Először is implementál két felsorolás típust, melyek inuitív módon leírják magukat:

```
type
  // Enum
  TInputType = (itClick, itKeyboard, itSpecialKey, itHotkey);
  TWaitType = (wtMil, wtSec, wtMin, wtHour);
```

Valamint definiálja a láncolt lista elemet és az arra hivatkozó mutatót:

```
  // Linked List pointer type
  PFlowElement = ^TFlowElement;

  // Linked List element
  TFlowElement = record
    inputType : TInputType;
    inputParam1 : string;
    inputParam2 : string;
    inputParam3 : string;
    inputParam4 : string;
    waitAfterAmount : integer;
    waitAfterType : TWaitType;
    waitAfterTypeText : string;
    deleteButton : TButton;
    panelObject : TPanel;
    labelObject : TLabel;
    NextElement : PFlowElement;
  end;
```

Ezek mellett még tartalmaz egy rutint, mely arra szolgál, hogy egy meglévő láncolt lista elemtől kezdve az összes további elemhez tartalmazó objektumot megfelelően szabadítsa fel a memóriából.

- **Unit_FlowHandler**: A folyamatokhoz tartozó legfonotsabb rutinokat definiálja, amik a következőek:

  1. Folyamat mentése fájlba,
  2. Folyamat betöltése állományból,
  3. Folyamat generálása a felhasználói bevitelből rögzített eseménysorból,
  4. Adott folyamati lépés végrehajtása, azaz input injektálása a Windows felé. pl.:

  ```
  if (currentStep.inputParam3 = 'Left') and (currentStep.
      inputParam4 = 'Down+Up (single)') then begin
    mouse_event(MOUSEEVENTF_LEFTDOWN, 0, 0, 0, 0);
    mouse_event(MOUSEEVENTF_LEFTUP, 0, 0, 0, 0);
  end
  ```

- **Unit_HookHandler**: Ennek az egységnek a feladata azoknak az objektumoknak, változóknak és függvényeknek az implementációja, melyek lehetővé teszik a felhasználói input rögzítését a Windows API *hook*-jainak felhasználásával.

  A *hook* egy olyan pont a rendszer üzenetkezelő mechanizmusában, ahová a szoftver egy olyan szubrutint telepít mely figyeli az üzenet forgalmat a rendszerben, és feldolgozza azokat még mielőtt elérné a cél-ablakhoz tartozó eljárását.

  A felhasználói input rözítésénél két ilyen *hook* kerül telepítésre, egy a billentyűzet üzenetsorának, a másik pedig az egérhez tartozóhoz. Ezek a telepítések a Windows API által bizotsított

  ```
  function SetWindowsHookEx; external user32 name '
      SetWindowsHookExW';
  ```

  függvény hívásával történnek.

  Ezeken túl az egység tartalmaz egy olyan funkciót is mely az adott konstanst (vagy karakterkódot) ember által könnyen értelmezhető szövegre fordítja, pl. **VK_PRIOR** → **[Page Up]**, vagy **65** → **[a]**

- **Unit_Status**: Ez az egység azt az ablakot implementálja, mely visszajelzést ad az éppen futó eseménysor-rögzítés lépéseiről.

  ```
  type
    TForm_Status = class(TForm)
      Lab_Input: TLabel;
      Lab_Finish: TLabel;
      Lab_Input_Title: TLabel;
      Lab_StepID_Title: TLabel;
      Pnl_Main: TPanel;
      Lab_StepID: TLabel;
      procedure FormCreate(Sender: TObject);
      procedure FormShow(Sender: TObject);
      procedure FormMouseMove(Sender: TObject; Shift: TShiftState;
          X, Y: Integer);
    public
  ```

```
    procedure UpdateLabel_Input(newText: string);
    procedure UpdateLabel_StepID(newID: integer);
  end;
```

A rutinjai lehetővé teszik, hogy más egységből (pl.: Unit_LinkedListHandler) is lehessen frissíteni a grafikus elemeit, valamint, hogy folyamatrögzítés közben hiába látszik az ablak, akkor se legyen útban a felhasználónak.

- **Unit_Scheduler**: Két egyszerű függvényt implementáló osztályt definiál, mely arra szolgál, hogy a Windows API segítségével meghívott *schtasks.exe* feladatütemezőbe rögzíteni, illetve onnan törölni lehessen folyamatokat.

```
  TScheduleHandler = class(TObject)
  public
    function DeleteTask(fPath: string): integer;
    function AddTask(fPath, sPath: string): integer;
  end;
```

- **Unit_DataGenerator**: Egy felületet és számos rutint definiál, melyek segítségével könnyedén folyamatokat lehet generálni. Ennek az a célja, hogy az adatbányászathoz elegendő mennyiségű folyamatot lehessen meghatározni emberi időn belül.

```
type
  TForm_Generator = class(TForm)
    Mem_Log: TMemo;
    Pnl_Interface: TPanel;
    Btn_Generate: TButton;
    RadGroup_GenCategory: TRadioGroup;
    Spin_GenCount: TSpinEdit;
    procedure Btn_GenerateClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose:
       Boolean);
  private
    { Private declarations }
    procedure Generate_ComputerShutdown(count: integer);
    procedure Generate_ComputerRestart(count: integer);
    procedure Generate_BrowserLaunch(count: integer);
    procedure AddToLog(msg: string);

    function GetClickDelay(_type: integer): integer;
    function GetRandomMouseCoordinate(min, max: integer):integer;
  end;
```

Három forgatókönyv került létrehozásra, ezekből választva lehetséges a generálás. A feladatot elvégezve a folyamatokat egy adott mappába állományonként menti le a szoftver.

- **Unit_Miner**: Egy felületet és az Alpha-algoritmust implementálja. A felületet használva a bányászat megkezdését követően láthatjuk, hogy éppen hol jár az algoritmustban a szoftver.

```
type
  TForm_Miner = class(TForm)
    Panel_Interface: TPanel;
    Mem_Log: TMemo;
    Edt_DataPath: TEdit;
```

```
    Pnl_DataPath: TPanel;
    Lab_DataPath: TLabel;
    Btn_DataPath_Browse: TButton;
    Btn_Begin: TButton;
    procedure FormCloseQuery(Sender: TObject; var CanClose:
        Boolean);
    procedure Btn_DataPath_BrowseClick(Sender: TObject);
    procedure Btn_BeginClick(Sender: TObject);
  private
    procedure AddToLog(msg: string);
    procedure AlphaMine();
    procedure ChangeUserControl(newState: boolean);
    function RemoveBracketsFromString(str: string): string;
    function IsInActivityList(str: string): boolean;
    function GetNewActivityID: string;
    function FindActivityID(str: string): string;
  end;
```

Amint végzett az algoritmus, az eredményeket a következő **Unit_MinerResults** által definiált ablakban ábrázolja.

A halmazpárok három-dimenziós adatstruktúraban vannak tárolva illetve kezelve, ezeknek az átlátása elég komoly odafigyelést igényel.

```
TArrayOfSets = array of array of array of integer;
```

- **Unit_MinerResults**: Azt a felületet implementáló egység, mely az Alpha-algoritmus eredményeit jelenít meg grafikusan. Ezek:

  1. A létrejött eseménynapló,
  2. Az eseménynaplóból meghatározott lenyomati mátrix,
  3. Az összes maximális halmaz,
  4. A kimeneti Petri-háló, melyet a **Unit_PetriHandler** segítségével generál.

```
TForm_MinerResults = class(TForm)
.
.
.
public
  procedure DrawPetriNet(arrayOfSets: TArrayOfSets);
  procedure DrawPlace(startX, startY: integer; lab: string);
  procedure DrawTransition(startX, startY: integer; lab: string
      );

  procedure DrawArrow(startX, startY, endX, endY: integer);
      overload;
  procedure DrawArrow(endX, endY: integer); overload;

  function GetStartEvents(): TIntegerArray;
  function GetEndEvents(): TIntegerArray;
end;
```

- **Unit_PetriHandler**: Számos objektumot és rutint implementál, melyek segítségével felépíthető és megjeleníthető egy Petri-háló

  1. Definiálja a helyeket,

```
TPetriPlace = record
  name: string;
  fromList: TStringArray;
  toList: TStringArray;
  location: TPoint;
  recursionLock: boolean;
end;
```

2. Definiálja az átmeneteket,

```
TPetriTransition = record
  id: integer;
  fromList: TStringArray;
  toList: TStringArray;
  location: TPoint;
  recursionLock: boolean;
end;
```

3. Definiál egy Petri-háló gyűjteményt, melyben tárolni lehet a helyeket és átmeneteket, valamint a rutinok segítségével fel lehet térképezni a közöttük lévő kapcsolatokat.

```
TPetriCollection = class(TObject)
  places: array of TPetriPlace;
  transitions: array of TPetriTransition;
  objectSize: integer;
public
  constructor Create();
  destructor Destroy(); override;
  procedure NewPlace(_name: string; _fromList, _toList:
     TStringArray);
  procedure NewTransition(_id: integer; _fromList, _toList:
      TStringArray);
  function FindIndexOfPlace(name: string): integer;
  function FindIndexOfTransition(id: integer): integer;
  procedure MapTransitions();
  procedure MapPlaceLocation(currentTransition:
     TPetriTransition);
  procedure MapTransitionLocation(currentPlace: TPetriPlace
     );
  procedure UpdateList(var list: TStringArray; newValue:
     string);
  function GetMaxIndexInColumn(col: integer): integer;
end;
```

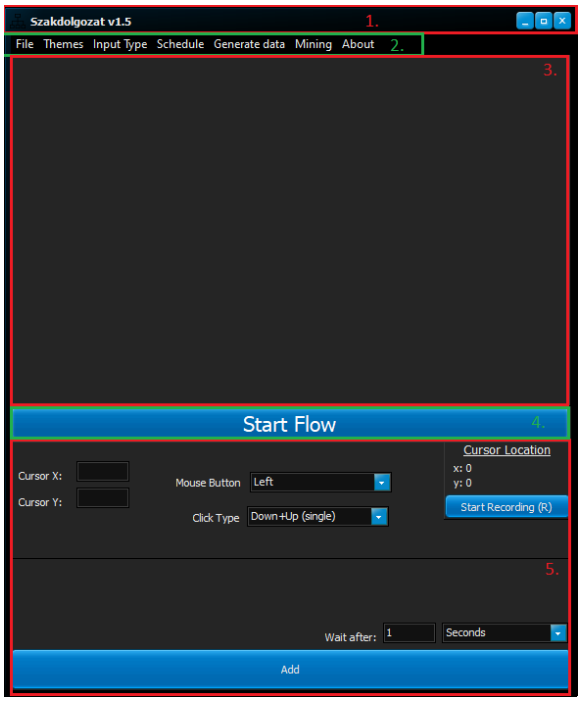
Indexek alapján kerülnek meghatározásra a helyek és átmenetek közötti kapcsolatok, így a megjelenített ábra balról-jobbra értelmezendő.

5. fejezet

Szoftverhasználat

A szoftver kezelőfelülete kifejezetten intuitívra lett tervezve, egyszerűen lehet vezérelni a programot, illetve megtalálni benne az egyes funkciókat.Az alábbi ábrán látható az a felület ami a program elindításakor nyílik meg.

5.1. ábra. Kezdeti felület



Magyarázat

1. Fejléc & rendszer menü
2. Főmenü
3. Folyamati panel
4. Folyamat indító gomb
5. Lépés hozzáadása - egér

5.1. Főmenü

Miután a program elindult, a különböző funkciók közötti navigálásra a főmenüt lehet használni. Ennek a használata az alábbiak alapján működik:

1. **File**: Itt van lehetőség a folyamatok külön fájlokként való kezelésére.

   Lehet:

   - Új folyamatot létrehozni,
   - Folyamatot fájlból betölteni,

5.2. ábra. "File" menü

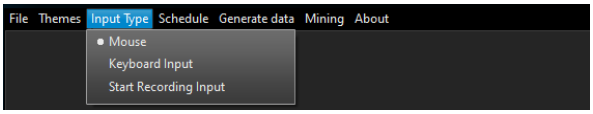
- Folyamatot lementeni állományba.

2. **Themes**: Ebben a menüben a szoftver felületének a megjelenítését lehet változtatni. Számos beépített témával rendelkezik amiből választani lehet a felhasználó kedvére.

5.3. ábra. "Themes" menü

3. **Input Type**: Van lehetőség a jelenlegi folyamathoz kézileg hozzáadni lépést, vagy hozzáfűzni olyan lépéseket amiket a program generál miután rögzítette a felhasználó eseménysorát. Ezeket a funkciókat érjuk el ezzel a menüvel.

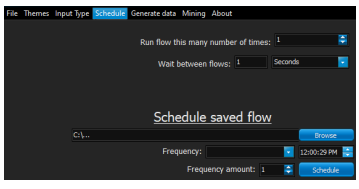
5.4. ábra. "Input type" menü



- **Mouse**: Egérkattintások hozzáadása kézileg
- **Keyboard Input**: Billentyű lenyomások hozzáadása kézileg
- **Start Recording Input**: Felhasználói eseménysor rögzítése, majd befejezés után lépések generálása.

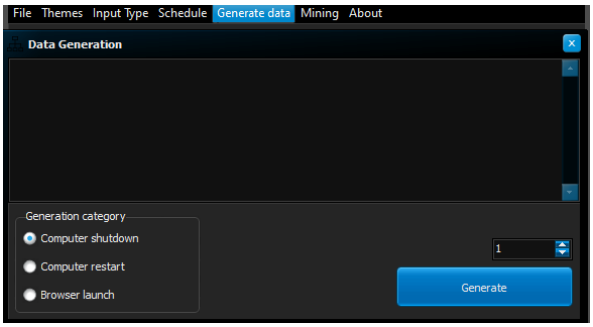
4. **Schedule**: Itt érhető el a folyamatok időzítésére szolgáló felület.

5.5. ábra. "Schedule" menü



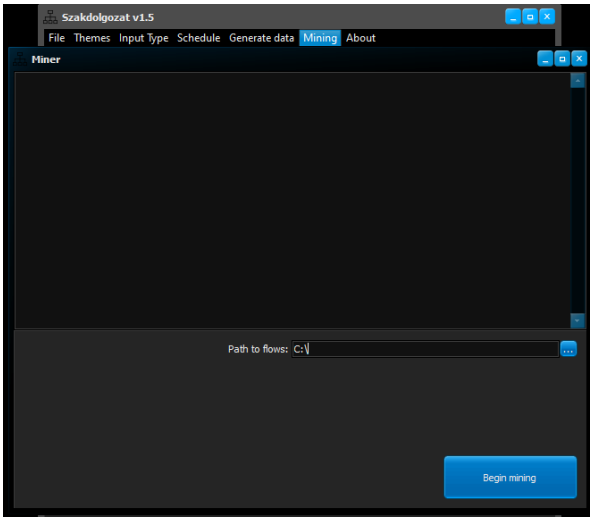
5. **Generate data**: Folymatokat lehet itt generálni előre meghatározott forgatókönyvek alapján.

5.6. ábra. "Generate Data" menü



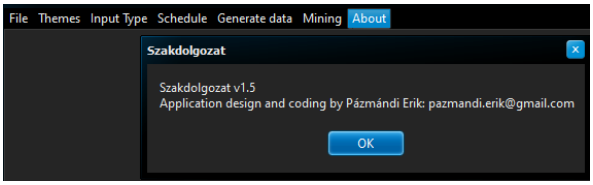
6. **Mining**: Az adatbányászatra szólgáló felületet itt érjük el.

5.7. ábra. "Mining" menü



7. **About**: Itt a készítői információ érhető el.

5.8. ábra. "About" menü

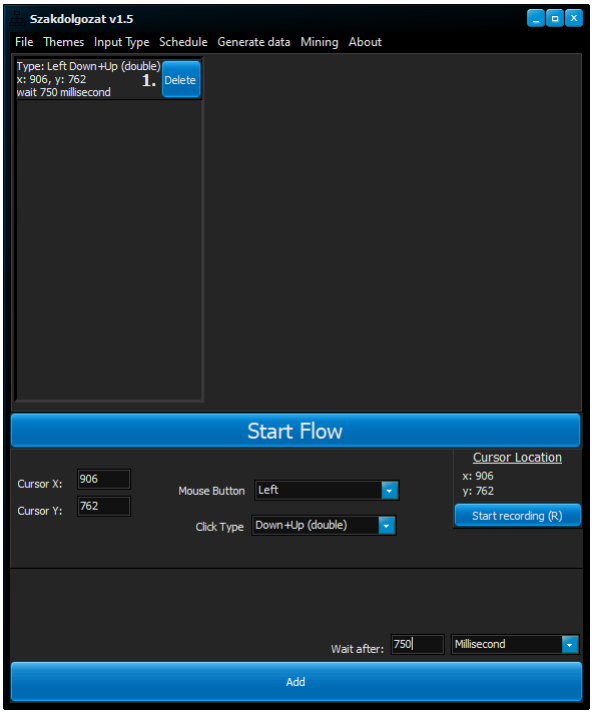


5.2. Folyamathoz tartozó vezérlés

A folyamatok vezérlésére több robosztus felületet biztosít a szoftver, ezeknek a segítségével lehet:

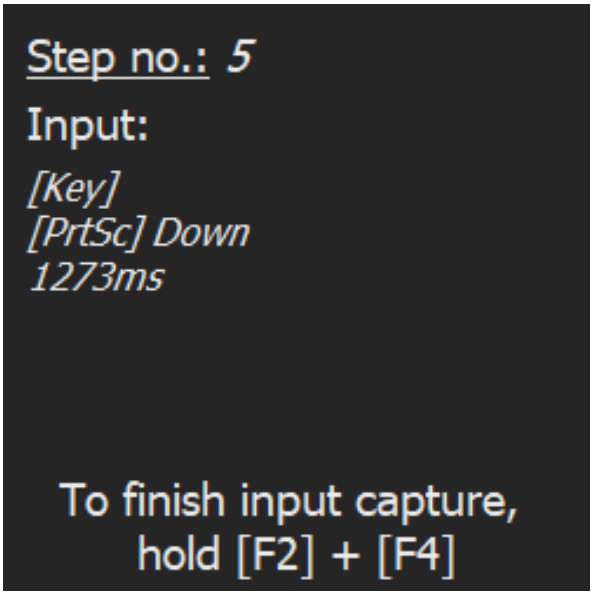
1. **Lépéseket kézileg hozzáadni**: A bevitel tipusának kiválasztása után meg lehet határozni a lépés paramétereit, majd az "Add" gombra kattintva hozzáadásra kerül a folyamathoz.

5.9. ábra. Lépés hozzáadása



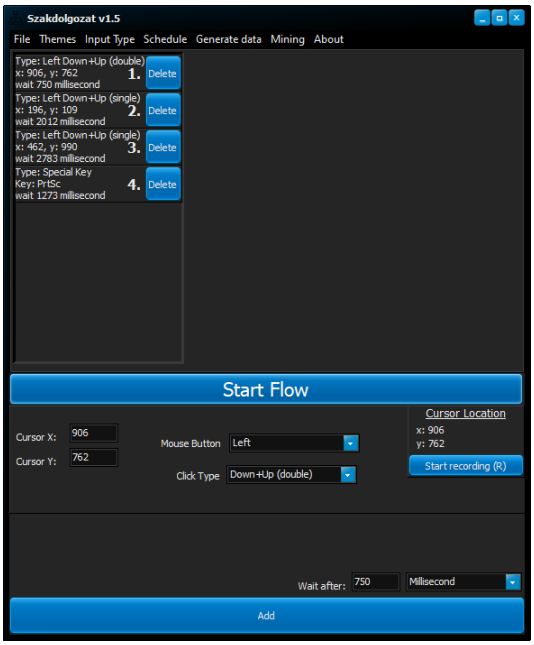
2. **Felhasználói eseménysort rögzíteni**: A funkció elindítása után a szoftver rögzíti a felhasználó által bevitt inputot, amiről visszajelzést biztosít egy külön ablakban.

5.10. ábra. Felhasználói eseménysor rögzítése



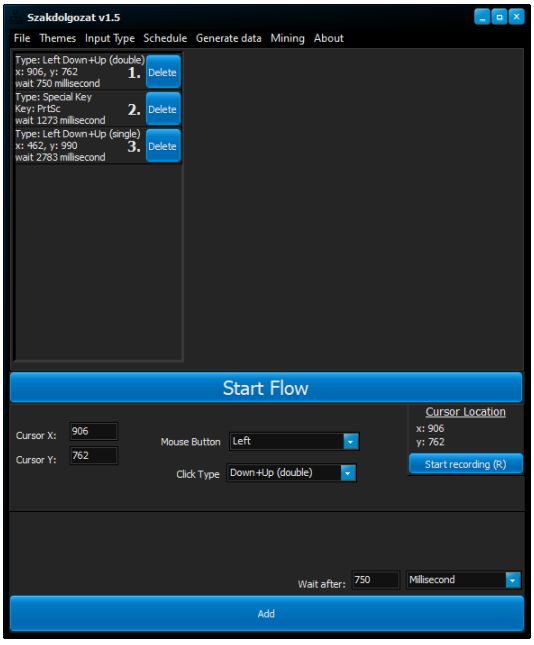
A funkció leállítását követően a rögzített lépesekből folyamati lépések kerülnek generálásra, amiket a szoftver fő felületén lehet látni.

5.11. ábra. Generált lépések



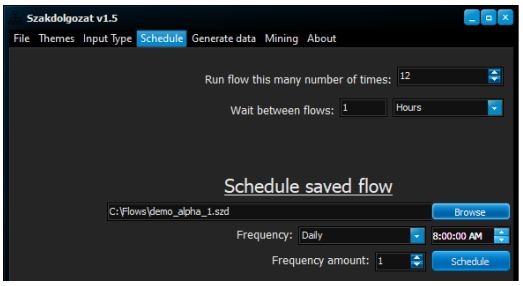
3. **Folyamatok lépéseit kezelni**: A "Delete" gombbal törölni lehet lépéseket, drag-and-drop stílusban pedig átrendezni őket.

5.12. ábra. Törlés és átrendezés



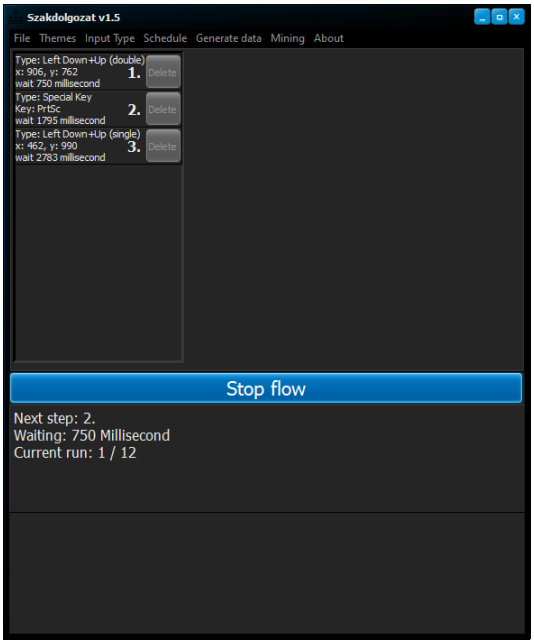
4. **Folyamatokat időzíteni**: Elkészített és lementett folyamatokat a beágyazott funkció segítségével lehet időzíteni.

5.13. ábra. Időzítés



5. **Folyamatokat visszajátszani**: A folyamatot elindítva az egyes lépések szekvenciálisan végrehajtódnak.

5.14. ábra. Futtatás

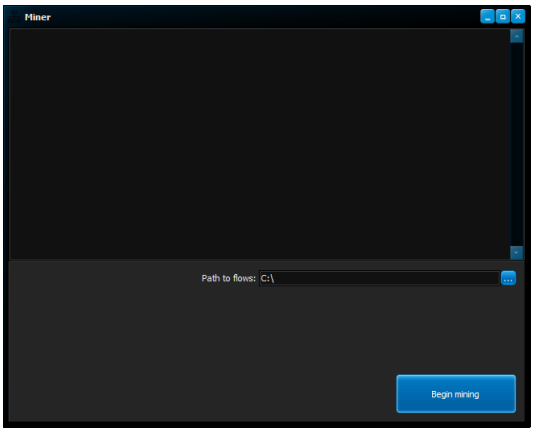


5.3. Adatbányászat

Az állományba lementett folyamatokon van lehetőség folyamatelemezés végrehajtására. Ehhez első lépésként össze kell gyűjteni az elemezni kívánt folymatokat egy könyvtárba.

Ezután a szoftver főmenüjében a "**Mining**" → "**Alpha miner**" útvonalon elérjük az adatbányászat kezdőfelületét.

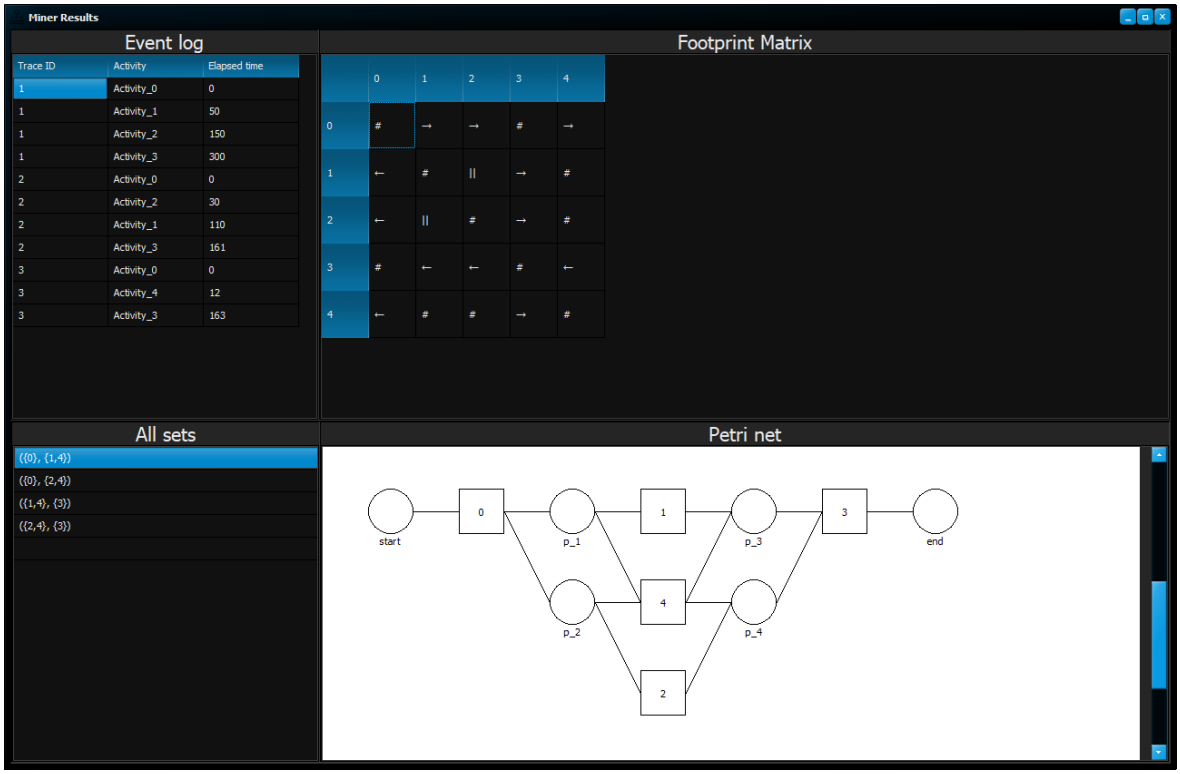
5.15. ábra. Adatbányászati felület



Ennek az ablaknak a tetején található a napló amiben az szerepel, hogy az adatbányászati folyamatban éppen mely lépésnél tart a szoftver.

A folyamatokat tartalmazó könyvtár kiválasztása után a "Begin mining" gombra kattintva elindul az adatbányászat. Miután végzett az algoritmussal a program, a bányászat eredményeit egy új ablakban vizualizálja.

5.16. ábra. Adatbányászati felület



6. fejezet

Összefoglalás

ELKÉSZÍTENI (1-2 oldal)

Irodalomjegyzék

Aalst, W., & Dongen, B. (2013, 01). Discovering petri nets from event logs. doi: 10.1007/978-3-642-38143-0_10

Alpha-algoritmus. (2022). *Alpha-algoritmus — Wikipedia, the free encyclopedia.* Retrieved from `https://en.wikipedia.org/wiki/Alpha_algorithm` ([Online, 2022-Október-07])

Embarcadero. (2022). *Delphi: Ide software overview.* Retrieved from `https://www.embarcadero.com/products/delphi` ([Online; 2022-Október-17])

Zarko, G. (2021). *Delphi history from pascal to embarcadero delphi xe 2.* Retrieved from `https://www.thoughtco.com/history-of-delphi-1056847`

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).