

SZAKDOLGOZAT



MISKOLCI EGYETEM

Rutinszerű feladatok automatizálása grafikus felhasználói felületek esetében

Készítette:

Pázmándi Erik

Programtervező informatikus

Témavezető:

Dr. Kovács Béla

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)

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 aláírással 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:

- szó szerinti 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. Bevezetés	1
2. Folyamatelemzés	2
2.0.1. Eseménynapló	2
2.1. Alpha-algoritmus	2
2.1.1. Grafikus ábrázolás	3
2.1.2. Példa	4
2.1.3. Korlátozások	6
2.2. Heurisztikus Bányász	6
2.2.1. Működése	6
2.2.2. Lépései	7
2.3. Robotic Process Automation	8
2.3.1. Alkalmazási területek	9
2.4. Delphi	9
2.4.1. Múltja röviden	10
2.4.2. Napjainkban	10
2.4.3. Delphi a dolgozathoz	11
3. Tervezés és Megvalósítás	12
3.1. A szoftver felépítése	12
3.2. Folyamatelemzés	13
3.2.1. Az Alpha-algoritmus	13
3.3. Implementáció	18
4. Szoftverhasználat	26
4.1. Főmenü	26
4.2. Folyamathoz tartozó vezérlés	29
4.3. Adatbányászat	34
5. Összefoglalás	36

1. fejezet

Bevezetés

Az automatizálás mindig is nagy szerepet játszott az emberiség életében, az információ korába lépve ennek fontossága magára az informatikára is kiterjed.

A dolgozat azt vizsgálja, hogy a számítógépek felhasználói felületén milyen sokszor elismételt folyamatok zajlanak le, ezek egy robot szempontjából hogyan is néznek ki, hogyan lehet ezeket utánózni, ezáltal automatizálni. Bemutatja a Robotikus Folyamat-automatizálás (RPA) koncepcióját, annak egy speciális esetét, valamint az ahhoz kötődő folyamatelemzési modellt.

Bár az RPA, mint fogalom annyira már nem új – a 2000-es évek elején bukkant felszínre először –, mégsem annyira elterjedt a köztudatban. Bár sokaknak félelmetes lehet, rengeteg szempontból ezt a technológiát lehet tekinteni az emberi erőforrás gépiesítése felé tett egyik korai lépésnek.

Számos előnnyel rendelkezik a technológia, például olcsóbb egy cég számára, mint egy munkavállaló felvétele aki ugyanazokat a tevékenységeket csinálná. Ezen túl megbízhatóbb is, hiszen az emberi munkással ellentétben nem unatkozik, nem fárad, így a hibalehetőségek aránya eredendően kisebb.

Ezeket túllépve, a dolgozat inkább egy alacsonyabb szinten tekinti át a technológiát, azaz bemutatja egy gyakorlati implementációját, annak használatát, valamint a felépített folyamatokon elvégzett elemzéseket.

Ezzel már egy képet fog kapni az Olvasó arról, hogy a hétköznapi életben előforduló folyamatokat hogyan tudja aránylag rövid időn belül automatizálni, teljes mértékben a saját kényel-mének megfelelően.

2. fejezet

Folyamatelemzés

A dolgozatban ismertetésre kerül két folyamatelemzési algoritmus, melyek célja, illetve számos paramétere átfedésben van. Ezeknek a célja, hogy eseménysorozatok halmazából egy ok-okozat rendszert építsen fel.

A működésükben az eseménysorok halmazát nevezhetjük eseménynaplónak is. Ez az eseménynapló úgynevezett trace halmazoknak a halmaza, egy trace pedig adott tevékenységnek a sorozata. Ezen fogalmak formális bevezetését láthatjuk a továbbiakban.

2.0.1. 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.

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

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

Számos gyakorlati megvalósításban a munkafolyamati naplót a munkafolyamattól függően át kell formázni, hogy egyértelműen tartalmazza a szükséges adatokat, majd az így kapott eredményt nevezzük eseménynaplónak.

2.1. Alpha-algoritmus

Az Alpha-bányász volt a legelső folyamatbányászati módszer amit javasoltak és egy egész jó rálátást biztosít a folyamatbányászat céljára, valamint arra, hogy a folyamatokban lévő különböző tevékenységek hogyan is hajtódnak végre. Emellett, az Alpha-bányász szolgáltat számos újabb folyamatbányászati technika (pl.: Heurisztikus bányász, genetikus bányászat) alapjaként. Először van der Aalst, Weijters és Märušter hozta be a köztudatba.

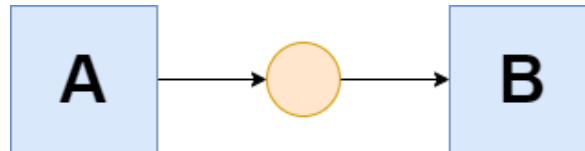
Az algoritmus egy munkafolyamati naplót $W \subseteq T^*$ kap bemenetként, és eredményként egy munkafolyamati hálót épít fel.

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.

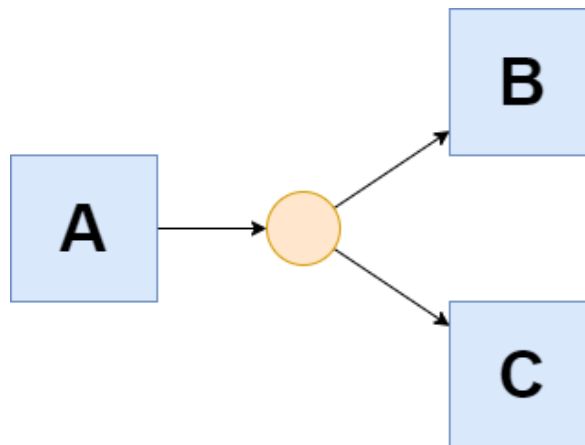
Az Alpha-bányász szabályai szerint az egyes tevékenységek között az alábbi 4 féle kapcsolat egyike lehetséges.

1. **Közvetlen sorrend:** $x > y$ akkor és csakis akkor ha az x eseményt közvetlenül követi y .
2. **Okozat:** $x \rightarrow y$ 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 $(y > x)$.

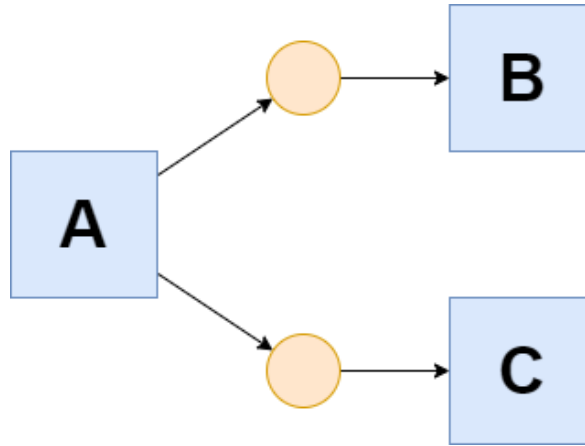
2.1.1. Grafikus ábrázolás



2.1. ábra. Szekvencia: $A \rightarrow B$



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

2.3. ábra. ÉS-elágazás: $A \rightarrow B$, $A \rightarrow C$ és $B \parallel C$

2.1.2. Példa

Vegyünk példának a 2.1. táblázatban látható eseménynaplót.

2.1. táblázat. Példa eseménynapló

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 modelljét.

Első lépésként létrehoz egy lenyomati mátrixot.

2.2. táblázat. Példa lenyomati mátrix

	A	B	C	D	E
A	#	\rightarrow	\rightarrow	#	\rightarrow
B	\leftarrow	#	\parallel	\rightarrow	#
C	\leftarrow	\parallel	#	\rightarrow	#
D	#	\leftarrow	\leftarrow	#	\leftarrow
E	\leftarrow	#	#	\rightarrow	#

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

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

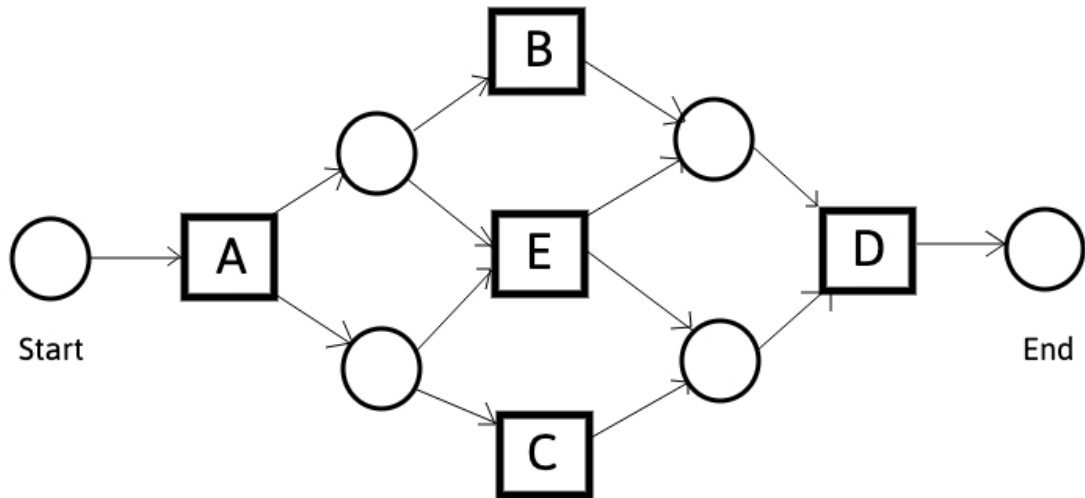
P_W tartalmazza az egyes Y_W -hez tartozó helyeket $p_{(A,B)}$, plusz a beviteli i_W helyet és a kimeneti o_W helyet. A folyamati reláció F_W az alábbiak uniójából áll össze:

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

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 munka-folyamati háló.

Ehhez a példához 2.4 ábrán látható Petri-háló jön létre az Alpha-bányász használatával.



2.4. ábra. Példa kimeneti Petri-háló

2.1.3. Korlátozások

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

[4]

2.2. Heurisztikus Bányász

Az Alpha-algoritmushoz hasonlóan ez is egy folyamatbányászati technika, mely lehetővé teszi eseménynaplókból információ kinyerését.

Ennek a sikerességéhez, az események sorrendjét csupán az egyes esetekben veszi figyelembe, azaz nem számít a sorrendjük esetek között.

2.2.1. Működése

Ahhoz, hogy egy folyamatmodellt meg lehessen találni egy eseménynapló alapján, azt elemezni szükséges. Találni kell okozati függőséget, például ha egy tevékenységet mindig követ egy másik adott tevékenység, akkor nagy valószínűséggel feltételezhető, hogy függőségi reláció van közöttük.

Ezeknek a relációknak az elemzéséhez vegyük az alábbiakat:

2.3. definíció. (*Heurisztikus bányász*): Legyen W egy eseménynapló adott T felett, azaz $W \subseteq T^*$. Emellett legyen $a, b \in T$:

1. $a >_W b$, ha léteznek $\sigma = t_1 t_2 t_3 \dots t_n$ trace és $i \in \{1, \dots, n-1\}$ úgy, hogy $\sigma \in W$ és $t_i = a$, valamint $t_{i+1} = b$,
2. $a \rightarrow_W b$, ha $a >_W b$ és $b \not>_W a$,
3. $a \#_W b$, ha $a \not>_W b$ és $b \not>_W a$,
4. $a \parallel_W b$, ha $a >_W b$ és $b >_W a$,
5. $a >>_W b$, ha létezik $\sigma = t_1 t_2 t_3 \dots t_n$ trace és $i \in \{1, \dots, n-2\}$ úgy, hogy $\sigma \in W$ és $t_i = a$, $t_{i+1} = b$ és $t_{i+2} = a$,
6. $a >>>_W b$, ha létezik $\sigma = t_1 t_2 t_3 \dots t_n$ trace, $i < j$ és $i, j \in \{1, \dots, n\}$ úgy, hogy $\sigma \in W$ és $t_i = a$, $t_j = b$.

2.2.2. Lépései

Első lépés: a függőségi gráf bányászata

A Heurisztikus Bányász első lépése az úgynevezett *függőségi gráf* felépítése. Ez egy gyakoriság alapú rendszer, mely azt mutatja meg, hogy mennyire lehetünk biztosak abban, hogy valóban léteznek függőségi reláció az egyes események között.

A kiszámított értékeket arra lehet használni egy heurisztikus keresésben, hogy megkapjuk a helyes függőségi relációt.

Legyen W egy eseménynapló adott T felett, valamint $a, b \in T$. Ekkor $|a >_W b|$ annak a száma, hogy $a >_W b$ mennyiszor fordul elő W -ben és az

$$a \Rightarrow_W b = \left(\frac{|a >_W b| - |b >_W a|}{|a >_W b| + |b >_W a| + 1} \right)$$

arány a függőség erősségét jelzi. Fontos megjegyezni, hogy ez alapján $a \Rightarrow_W b$ értéke mindig -1 és 1 között van. Egy magas érték alapján erősen feltételezhető, hogy létezik a függőségi reláció.

Második lépés: a ciklusok felismerése

Egy folyamatban előfordulhat, hogy egy adott tevékenység többször kerül végrehajtásra. Ez tipikusan ciklusnak felel meg az adott modellben.

A távoli ciklusok (pl.: ...ABCABCABC...) nem jelentenek problémát a Heurisztikus bányásznak, azonban az egyes hosszúságú (pl.: ACB, ACCB, ACCCB, ... tracekben) és kettes hosszúságú (pl. olyan tracek mint: ACDB, ACDCDB, ACDCDCDB) ciklusoknál a $C \Rightarrow_W C$ és $C \Rightarrow_W D$ értéke mindig nagyon alacsony.

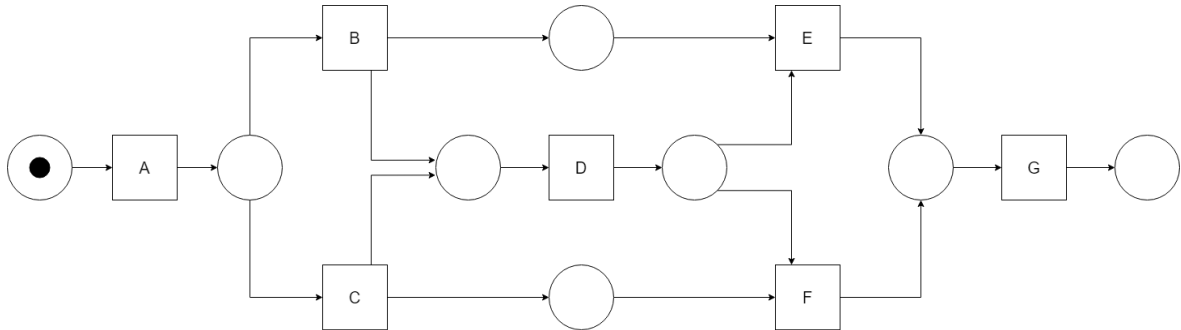
Szerencsére nagyon egyszerű módon lehet definiálni a függőség mértékét ezekre a ciklusokra. Legyen W egy eseménynapló adott T felett, valamint $a, b \in T$. Ekkor $|a >_W b|$ annak a száma, hogy $a >_W b$ mennyiszor fordul elő W -ben, illetve $|a >>_W a|$ annak a száma, hogy $a >>_W b$ mennyiszor fordul elő W -ben, ahol

$$a \Rightarrow_W a = \left(\frac{|a >_W a|}{|a >_W a| + 1} \right),$$

$$a \Rightarrow_2 b = \left(\frac{|a >>_W b| + |b >>_W a|}{|a >>_W b| + |b >>_W a| + 1} \right).$$

Harmadik lépés: a távoli függőségek feltérképezése

Néhány folyamatmodellben, két tevékenység közül történő választás nem feltétlenül egy adott lépésben dől el, hanem a folyamatmodell más részein történő választásoktól függhet.



2.5. ábra. Folyamatmodell távoli függőséggel

A 2.5-ös ábrán látható egy távoli függőség felépítése. D tevékenység végrehajtása után E és F tevékenységek közül lehet választani. Azonban ezt az E és F közötti választást egy korábbi, B és C közötti választás előzi meg. Egyértelműen, ezeket a nem-lokális viselkedéseket kifejezetten nehéz bányászni az olyan megközelítésekkel melyek bináris információ alapján ($a >_W b$) működnek. Csupán néhány algoritmus képes sikeresen bányászni őket.

Azt az eseménynaplót amelyet a 2.5-ös ábra folyamatmodellje alapján kapunk, a Heurisztikus Bányász eddig bemutatott állapotában úgy fogja elemezni, hogy a függőségi gráfban nem fog szerepelni a $B \rightarrow E$ és $C \rightarrow F$ kapcsolat. Azonban az említett $a >>>_W b$ reláció erősen jelezni fogja, hogy B -t mindig követi E , C -t pedig F . Például ha $|B|$ a gyakorisága B -nek, akkor $B \Rightarrow_W^l E = |B| >>>_W E|/|B| + 1$ értéke 1-hez közeli lesz, azonban sok magas \Rightarrow_W^l érték már összhangban van a folyamatmodellel. Ennek megfelelően, a 2.5-ös ábra alapján az $A \Rightarrow_W^l D$ értéke közel lesz 1-hez, viszont nincsen szükség a plusz függőségi relációra.

Ezt az alapján lehet ellenőrizni, hogy az eddig bányászott folyamatmodellel leseteljük, hogy lehetséges-e A -ból a végállapotba (G) eljutni anélkül, hogy áthaladnánk D tevékenységen. Csakis akkor, ha ez lehetséges, akkor kerül a folyamatmodell frissítésre az extra A -ból D -be függőségi relációval. [5]

2.3. 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.

2.3.1. Alkalmazási területek

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

Elsősorban az adott cégtől függ, hogy belevág-e egy ilyen szoftvertechnológias megoldásba, a következőkben szerepel 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 automatizálniak tekinteni.

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

- UIPath,
- Microsoft Power Automate,
- Blue Prism,
- Automaton Anywhere,
- Kofax.

2.4. Delphi

A dolgozathoz készült szoftver Delphi nyelven íródott, így fontos legalább nagyvonalakban ismerni a nyelvet, hogy megértsük a választás okát. 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 fordítói natív kódot generálnak a célrendszerrel függően, legyen az Microsoft Windows, macOS, iOS, Android vagy Linux (x64).^[3]

2.4.1. Múltja röviden

Az „anyanyelve” a Delphine a Pascal, ami pedig a modellje nagy részét az Algolnak köszönheti – az első magasszintű programozási 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 épülő 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 leányvállalatot 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 Embarcadero márka alatt működteti a fejlesztői eszközök divízióját.

Az évek alatt rendkívül sok modernizáción ment át a Delphi. OLE (Object Linking and Embedding) automatizáció és változó adattípus támogatásától kezdve, DLL debugoláson és XML támogatáson keresztül egészen a multi-platform alkalmazásokig és az in-line változó deklarálásig.[6]

2.4.2. 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 ismertségre 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-platformatás:** 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 előnyös, mert semmilyen egyéb keretrendszer telepítésére nincsen szüksége (pl.: .NET keretrendszer, Java Runtime Environment)
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 felgyorsítva magát a fejlesztési és hibakeresési folyamatokat.

2.4.3. 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 kapcsolatban 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 és Megvalósítás

A dolgozat által vizsgált témához 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.

3.1. A szoftver felépítése

A legmagasabb szinten a 3.1. ábra nyújtja a legtisztább áttekintését a különböző funkcióknak és a szoftver sokszínűségének.

Felhasználói interfész					
Különböző, felhasználói felülethez kapcsolódó szálak	Láncolt lista kezelő	Folyamat rögzítő	Folyamat generáló	Adatbányász	Eredmény ábrázoló
		Folyamat lejátszó	Folyamat időzítő		Petri háló kezelő
Kiegészítő funkciótár			Konfiguráció kezelő		

3.1. ábra. Szerkezeti 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.

- **Folyamat 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 hozzá tartozó függvényeket a szoftver számára értelmezhető módon implementálja.
- **Kiegészítő funkció táár:** Számos hasznos funkció gyűjteménye, melyet a többi alrendszer használ.
- **Konfiguráció kezelő:** Futásidők között felhasználói preferenciák és beállítások tárolásáért és betöltéséért felel.

3.2. Folyamatelemzés

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.

3.2.1. Az Alpha-algoritmus

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.

$$E_L = \{e \in E \mid \exists \sigma \in L, e \in \sigma\},$$

2. Definiáljuk az összes bemeneti eseményt.

$$E_I = \{e \in E \mid \exists \sigma \in L, e = first(\sigma)\},$$

3. Definiáljuk az összes kimeneti eseményt.

$$E_O = \{e \in E \mid \exists \sigma \in L, e = last(\sigma)\},$$

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.

$$X_L = \{(A, B) \mid A \subseteq E_L \wedge A \neq \emptyset \wedge B \subseteq E_L \wedge B \neq \emptyset \wedge \\ \forall_{a \in A} \forall_{b \in B} a \rightarrow_L b \wedge \forall_{a_1, a_2 \in A} a_1 \#_L a_2 \wedge \forall_{b_1, b_2 \in B} b_1 \#_L b_2\},$$

5. Elhagyjuk a nem-maximiális halmazokat.

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

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

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

7. Berajzoljuk a kapcsolatokat.

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

8. Visszatérünk a Petri-hálóval.

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

[1]

3.2. példa. Itt három előre létrehozott folyamaton kerül alkalmazásra az 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, részeredményeként eseménynapló és lenyomati mátrix jön létre, kimentként pedig egy olyan Petri-háló kerül generálásra mely leírja a folyamat modelljét.

Vegyük a 3.1. táblázatot.

3.1. táblázat. 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 feltüntetve,
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 a 3.2. táblázatban látható eseménynaplót kapjuk.

3.2. táblázat. 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. $E_I = \langle T_0 \rangle$,
2. $E_O = \langle T_3 \rangle$.

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

Ezzel jön létre a 3.3. táblázatban látható lenyomati mátrix.

3.3. táblázat. Lenyomati mátrix

	T_0	T_1	T_2	T_3	T_4
T_0	#	\rightarrow	\rightarrow	#	\rightarrow
T_1	\leftarrow	#	\parallel	\rightarrow	#
T_2	\leftarrow	\parallel	#	\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 a 3.4. táblázatban látható halmazpárok lehetségesek a jelenlegi példában:

3.4. táblázat. Lehetséges halmazpárok

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

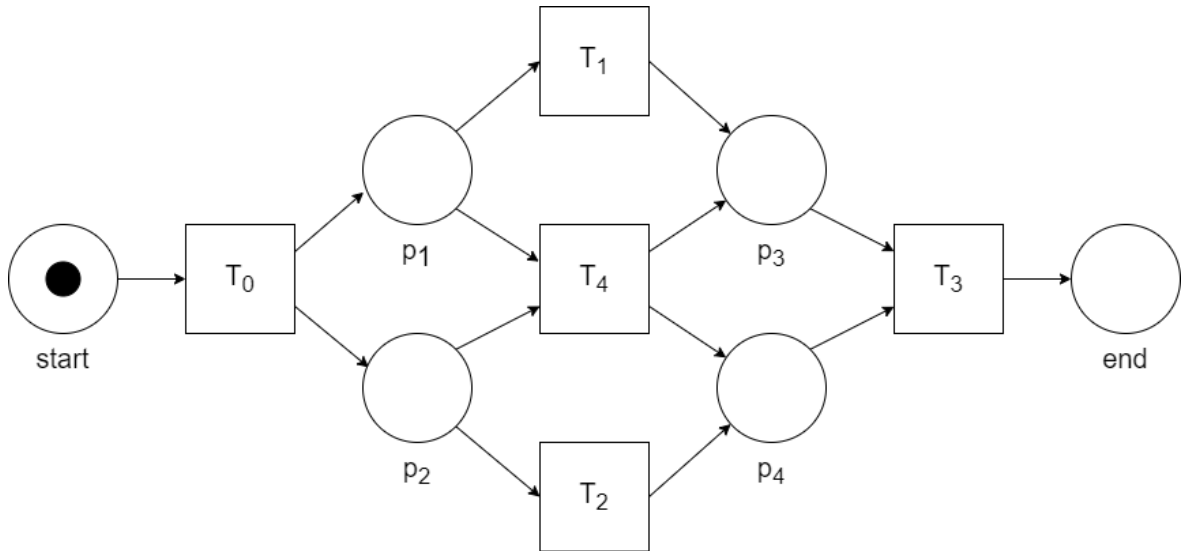
Következő lépésként ezekből a halmazpárokból kell eltávolítani a nem-maximálisakat, azaz azokat amik részhalmazai egy másiknak. Ezt program szinten egy többszörös ciklus segítségével könnyedén el lehet végezni, jelen példában pedig az 3.5. táblázatban található, négy halmazpár maradt:

3.5. táblázat. 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.



3.2. ábra. Kimeneti Petri-háló

[2]

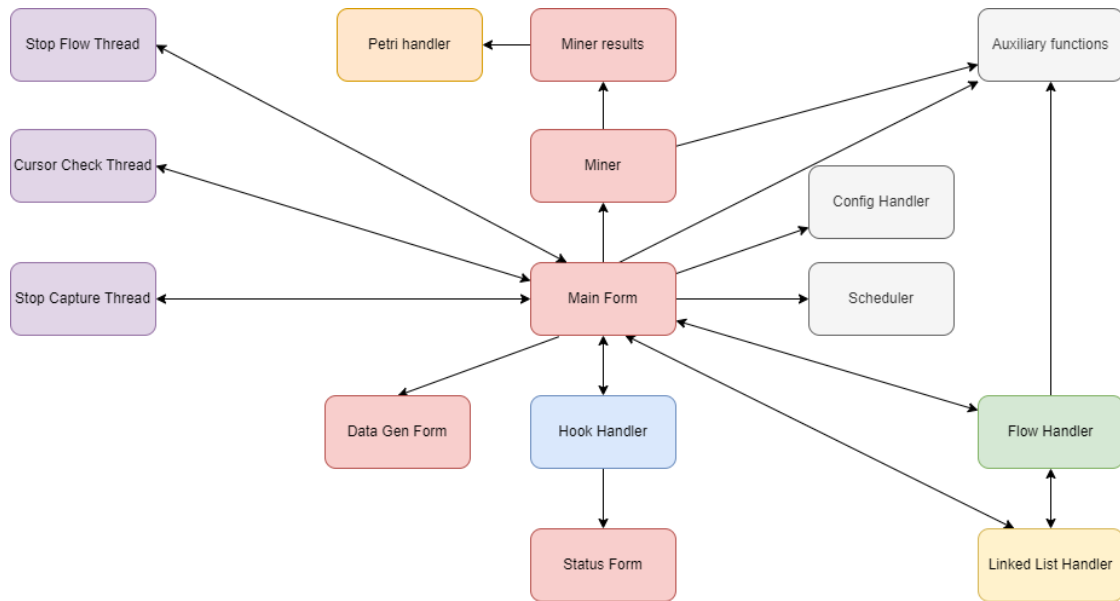
3.3. Implementáció

A program gyakorlati megvalósítása egy-két apróságtól eltekintve megegyezik a tervzettel. Fejlesztés közben felmerültek ötletek, melyek javítottak az eredeti terven, de az alapvető struktúra megmaradt, az egymáshoz tartozó funkciók, változók és kódrészletek pedig külön egységekbe (továbbiakban: unitok) lettek szervezve, így csoportosítva az egyes szoftverfunkciókat.

Ezek a unitok többféle kapcsolatban állhatnak egymással, az egymásra való hivatkozásuknak függvényében. Ezeket a kapcsolatokat a 3.3 ábra demonstrálja.

Pirossal azok a unitok láthatók melyekhez tartozik grafikus felület is, a további színek pedig az egyes funkciók csoportosítására szolgálnak. Részletesebben a unitokról a következő pontokban olvashatunk.

- **Unit_Main:** Ez a program főegysége, ebben került implementálásra a főablak és annak felhasználói felülete. Az alábbi feladatokat látja 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:** A konfigurációkezelő osztályt, az abba tartozó objektumokat és rutinokat implementálja, melyek lehetővé teszik a futásidő alatti konfigurációs beállítások titkosítva történő tárolását.



3.3. ábra. Unitok kapcsolatai

```

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 szálát 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 kézzel akar egérekattintást hozzáadni az aktuális folyamathoz, akkor megkönnyítse a megfelelő képernyő-koordináták meghatározását.

- **Unit_StopCaptureThread:** Egy olyan szálát 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 esemény sor rögzítését.

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

Először is implementál két felsorolás típust, melyek intuitív 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 tartozó objektumot megfelelően szabadítsa fel a memóriából.

- **Unit _FlowHandler:** A folyamatokhoz tartozó legfontosabb rutinokat definiálja, amik a következők:

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 rögzítésénél két ilyen *hook* kerül telepítésre: egy a billentyűzet üzenetsornak, a másik pedig az egérhez tartozóhoz. Ezek a telepítések a Windows API által biztosított

```

function SetWindowsHookEx; external user32 name '
SetWindowsHookExW';

```

függvény hívásával történnek.

Ezekén 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_HookHandler**) 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 adatbázáshoz 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 forгатókönyv került létrehozásra, ezekből választva lehetséges a generálás. A feladatot elvégezve a folyamatokat egy adott mappába állományonként menti le a szoftver.

- **Unit_Miner:** Egy felület ami az Alpha-algoritmust illetve a Heurisztikus Bányászt implementálja. A felületet használva a bányászat megkezdését követően láthatjuk, hogy éppen hol jár a választott 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 adatstruktúrában vannak tárolva, illetve kezelve. Ezeknek az értelmezése elég komoly odafigyelést igényel.

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

- **Unit_MinerResults:** Azt a felületet implementáló egység, mely a választott bányász eredményeit jeleníti meg grafikusán. Ezek:

1. a létrejött eseménynapló,
2. az eseménynaplóból létrejött lenyomati mátrix (Heurisztikus bányásznál függőségi mátrix),
3. az összes maximális halmaz (Alpha-algoritmus esetében),
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;

```

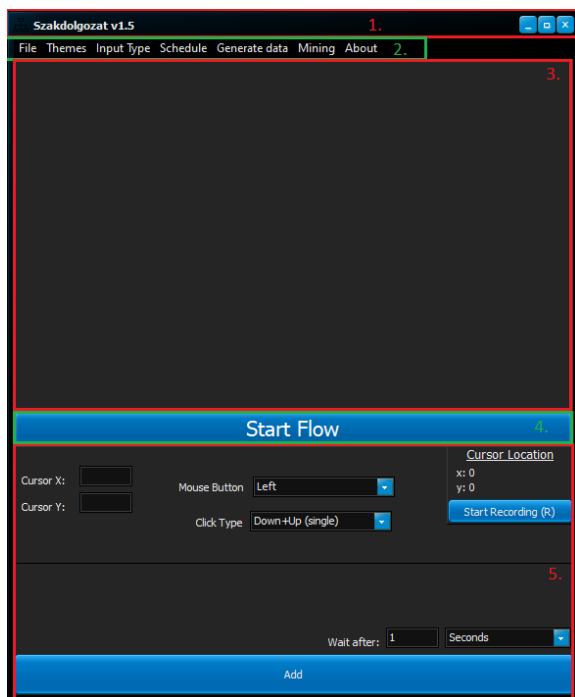
`end;`

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

4. fejezet

Szoftverhasználat

A szoftver kezelőfelületének tervezésénél kiemelt szempont volt, hogy intuitív legyen, egyszerűen lehessen vezérelni a programot, illetve megtalálni benne az egyes funkciókat. A 4.1 ábrán látható az a felület, ami a program elindításakor nyílik meg.



4.1. ábra. Kezdeti felület

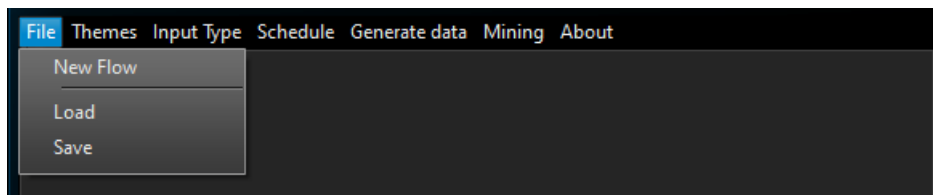
Magyarázat

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

4.1. Főmenü

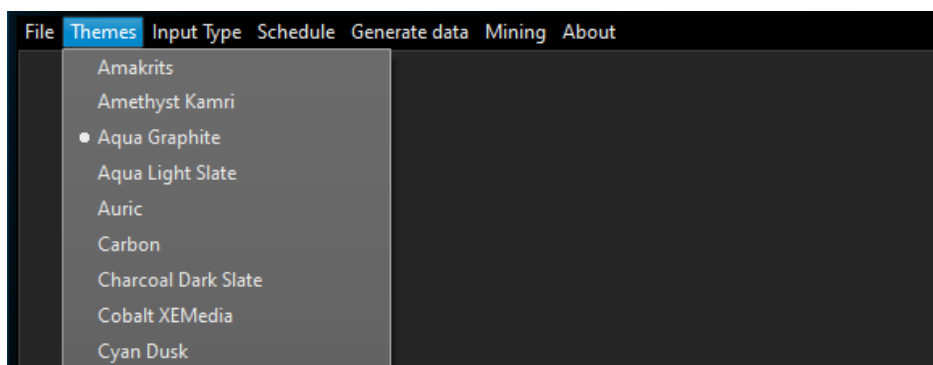
Miután a program elindult, a különböző funkciók közötti navigálásra a főmenüt lehet használni. Ennek a használata az alábbi pontokban kerül áttekintésre.

1. **File:** Itt van lehetőség a folyamatok külön fájlként való kezelésére (4.2. ábra). Lehet:
 - új folyamatot létrehozni,
 - folyamatot fájlból betölteni,
 - folyamatot lementeni állományba.



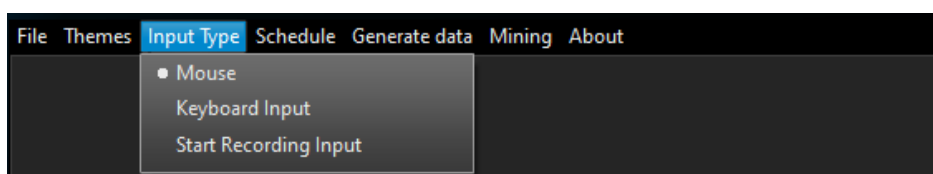
4.2. ábra. "File" menü

2. **Themes:** Ebben a menüben a szoftver felületének a megjelenítését lehet változtatni. Számos beépített témával rendelkezik, amiből választhat a felhasználó kedvére (4.3. ábra).



4.3. ábra. "Themes" menü

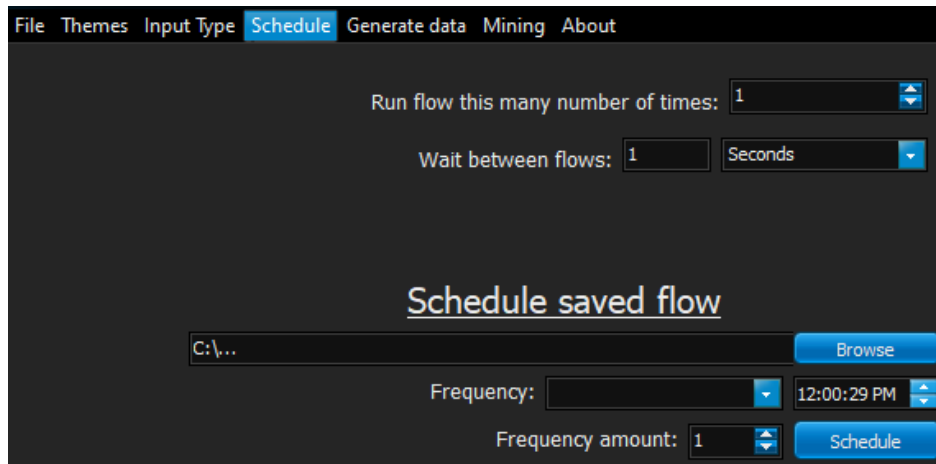
3. **Input Type:** Van lehetőség a jelenlegi folyamathoz kézzel hozzáadni lépést, vagy hozzáfűzni olyan lépéseket, amiket a program generál miután rögzítette a felhasználó eseménysorát. Ezeket a funkciókat érjük el ezzel a menüvel (4.4. ábra).



4.4. ábra. "Input type" menü

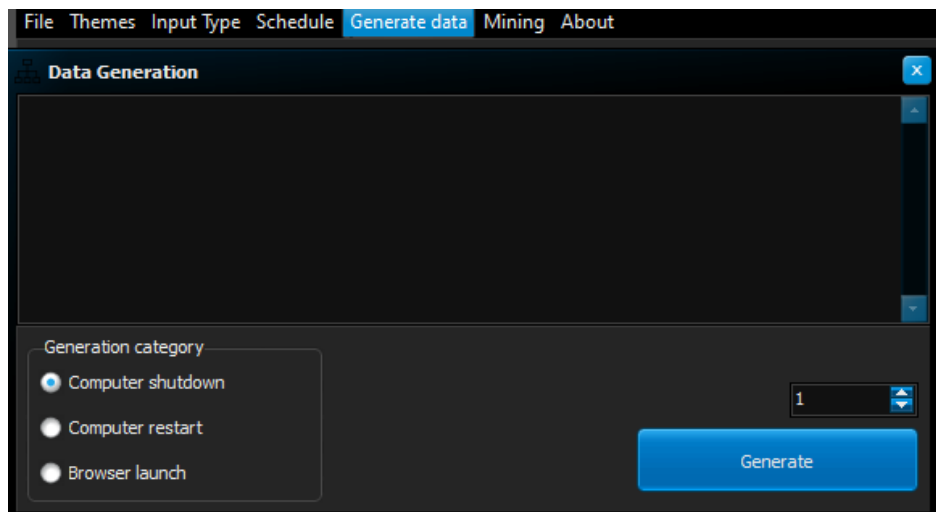
- **Mouse:** Egérkattintások hozzáadása kézzel.
- **Keyboard Input:** Billentyű lenyomások hozzáadása kézzel.
- **Start Recording Input:** Felhasználói eseménysor rögzítése, majd befejezés után lépések generálása.

4. **Schedule:** Itt érhető el a folyamatok időzítésére szolgáló felület (4.5. ábra).



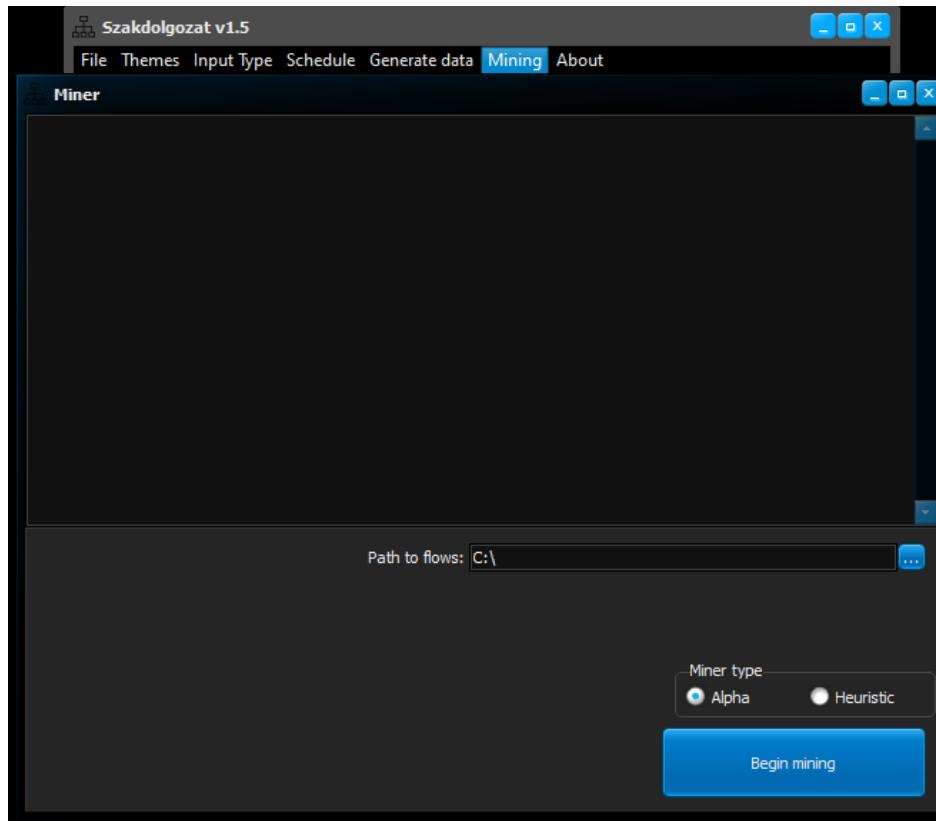
4.5. ábra. "Schedule" menü

5. **Generate data:** Folyamatokat lehet itt generálni előre meghatározott forgatókönyvek alapján (4.6. ábra).



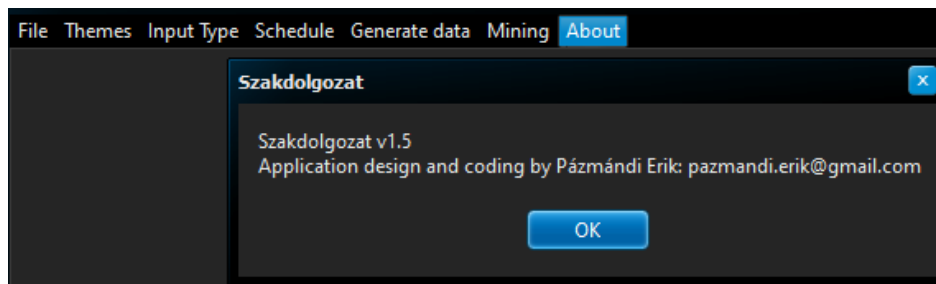
4.6. ábra. "Generate Data" menü

6. **Mining:** Az adatbányászatra szolgáló felületet itt érjük el (4.7. ábra).



4.7. ábra. "Mining" menü

7. **About:** Itt a készítői információ érhető el (4.8. ábra).

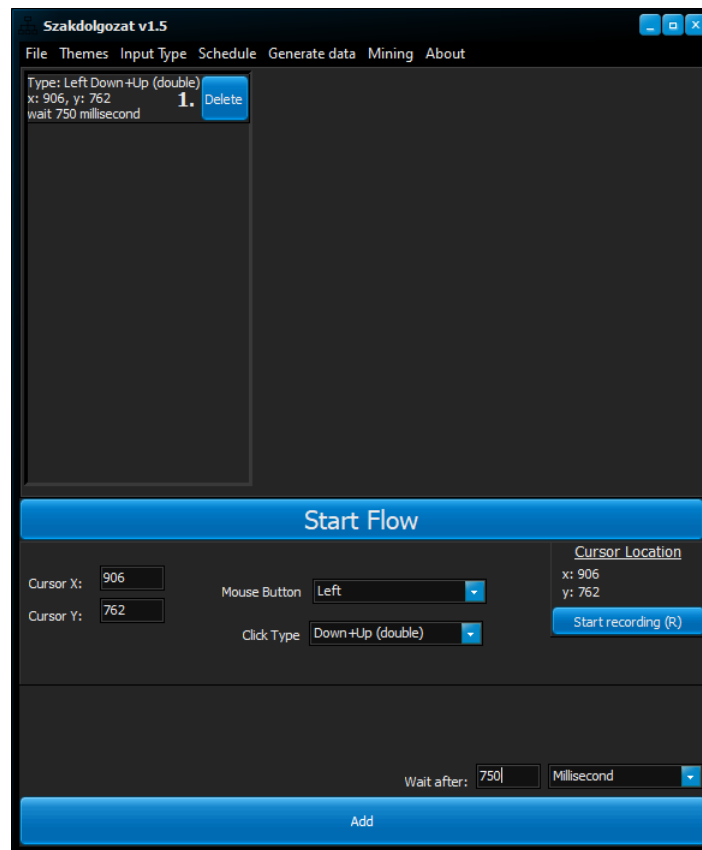


4.8. ábra. "About" menü

4.2. Folyamathoz tartozó vezérlés

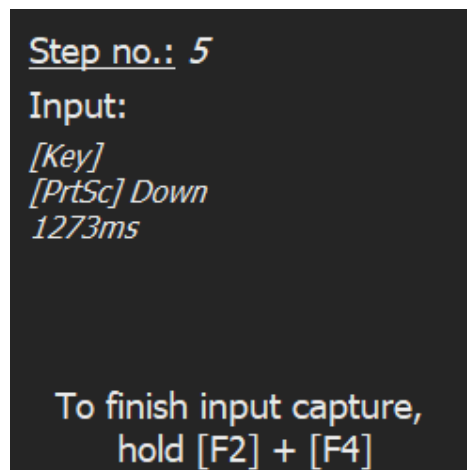
A folyamatok vezérlésére több robusztus felületet biztosít a szoftver, ezeknek a segítségével lehet:

1. **Lépéseket kézzel hozzáadni:** A bevitel típusának kiválasztása után meg lehet határozni a lépés paramétereit, majd az "Add" gombra kattintva hozzáadásra kerül a folyamathoz (4.9. ábra).



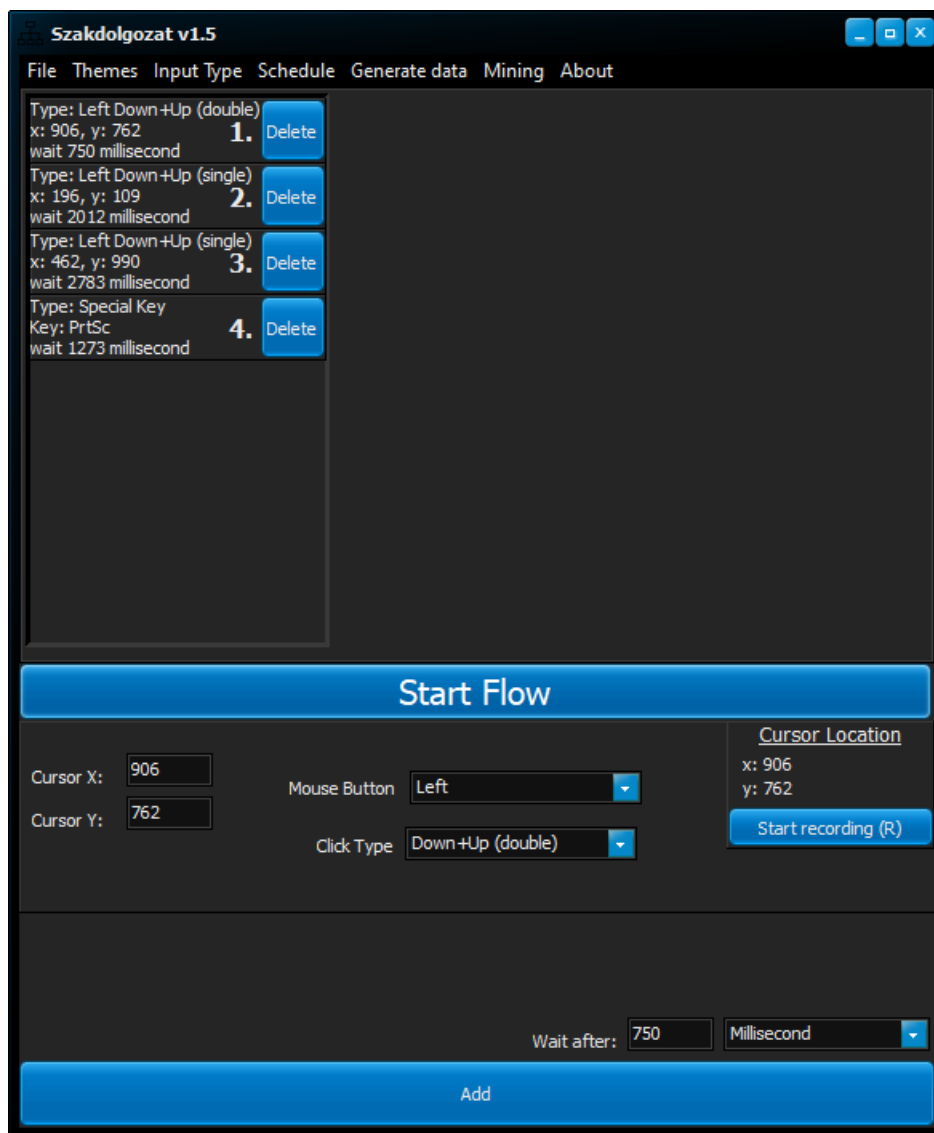
4.9. ábra. Lépés hozzáadása

2. **Felhasználói eseménysort rögzíteni:** A funkció elindítása után a szoftver rögzíti a felhasználó által bevitt inputot, amiről visszajelzést biztosít egy külön ablakban. (4.10. ábra).



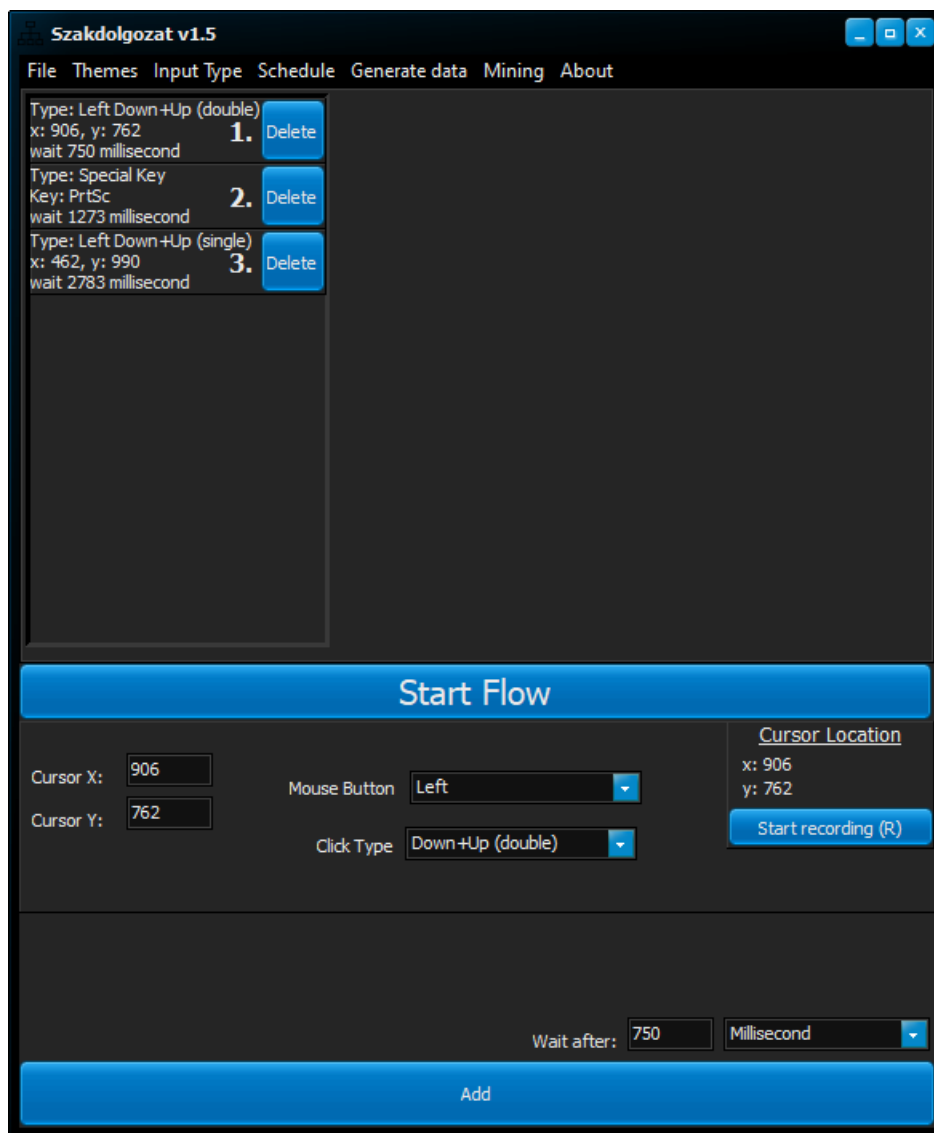
4.10. ábra. Felhasználói eseménysor rögzítése

A funkció leállítását követően a rögzített lépésekből folyamati lépések kerülnek generálásra, amiket a szoftver fő felületén lehet látni (4.11. ábra).



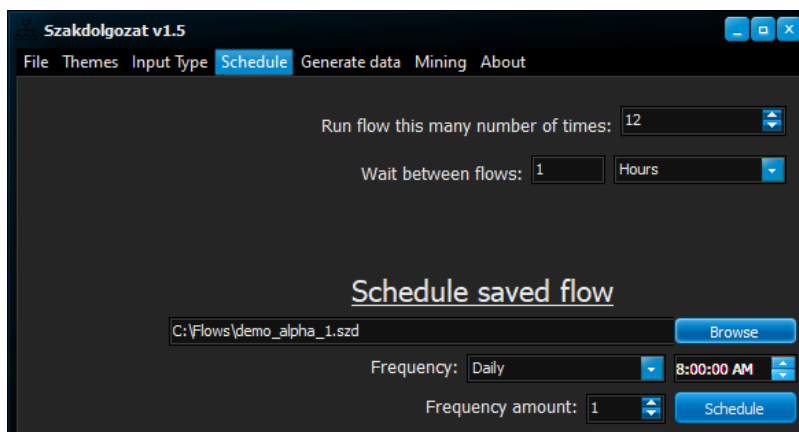
4.11. ábra. Generált lépések

3. **Folyamatok lépéseit kezelni:** A "Delete" gombbal törölni lehet lépéseket, drag-and-drop stílusban pedig átrendezni őket (4.12. ábra).



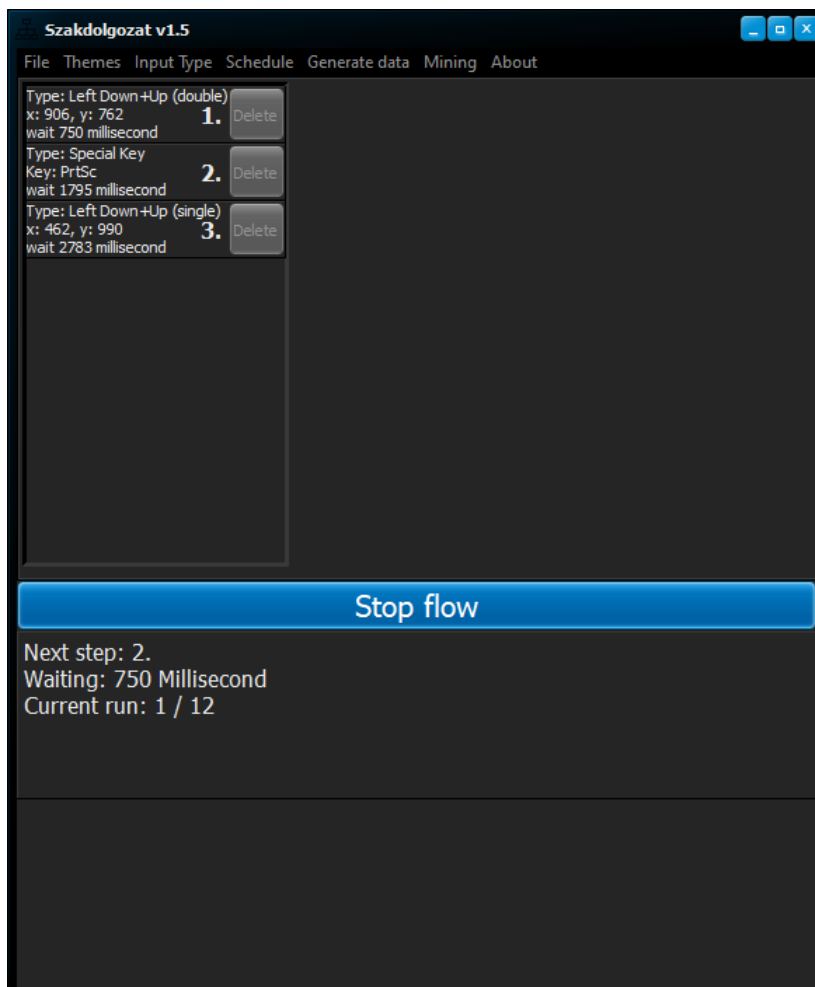
4.12. ábra. Törlés és átrendezés

4. **Folyamatokat időzíteni:** Elkészített és lementett folyamatokat a beágyazott funkció segítségével lehet időzíteni (4.13. ábra).



4.13. ábra. Időzítés

5. **Folyamatokat visszajátszani:** A folyamatot elindítva az egyes lépések szekvenciálisan végrehajtnak (4.14. ábra).

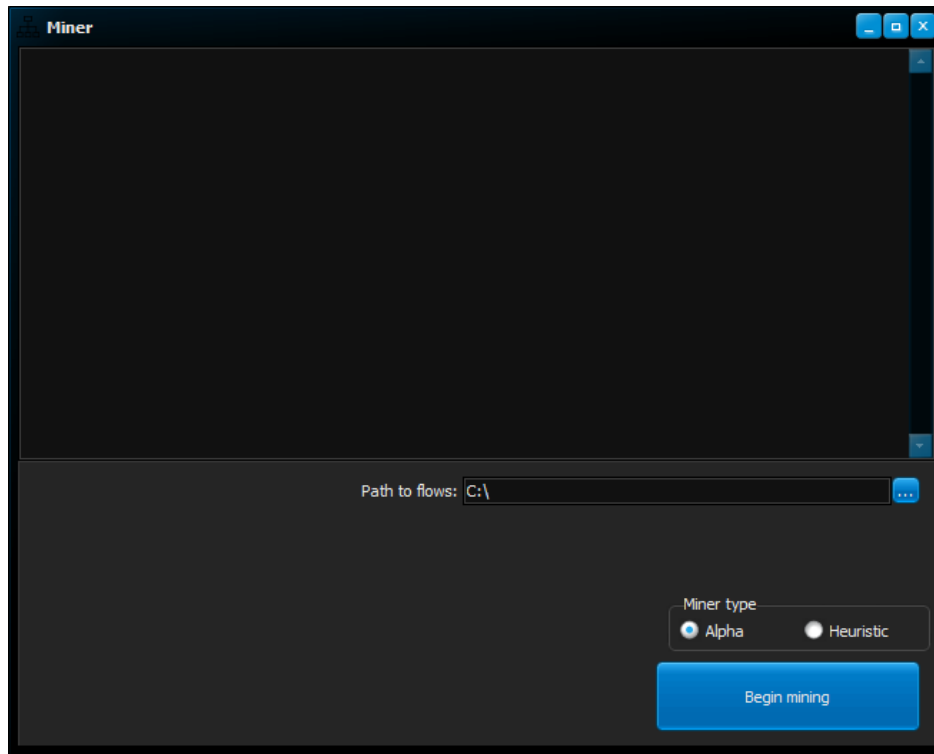


4.14. ábra. Futtatás

4.3. Adatbányászat

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

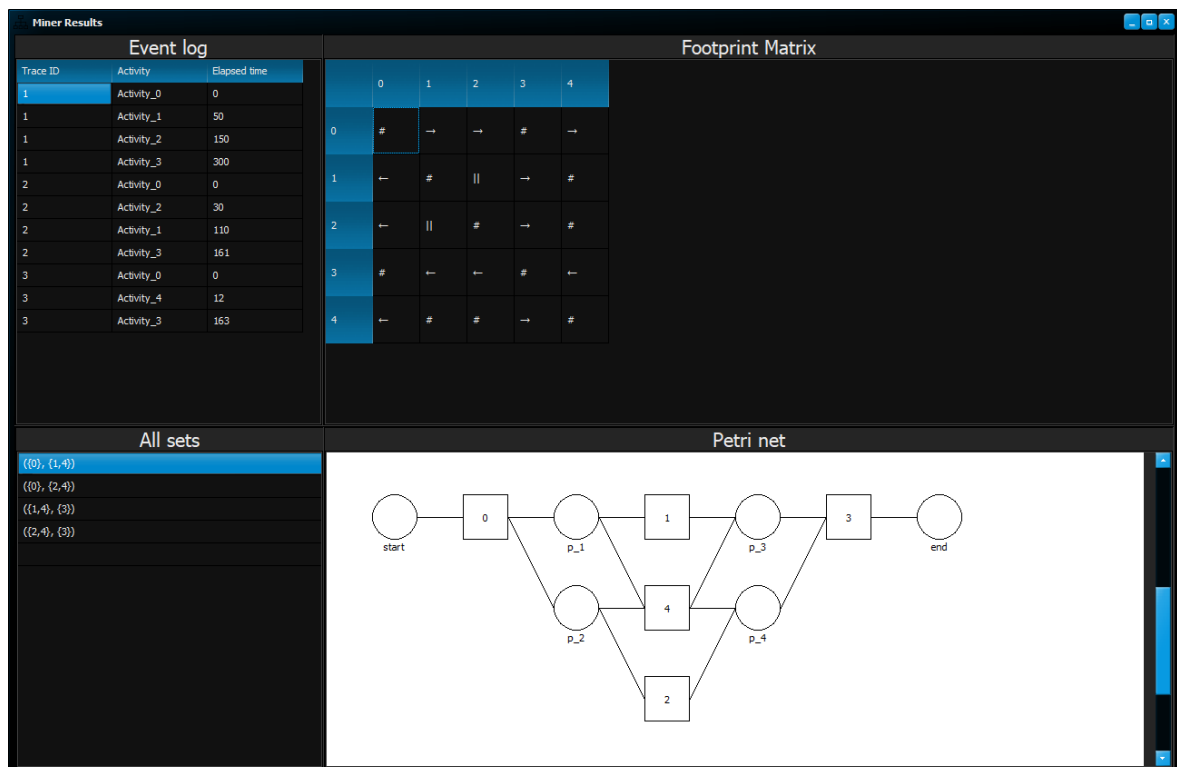
Ezután a szoftver főmenüjében a **"Mining"** → **"Alpha miner"** útvonalon elérjük az adatbányászat kezdőfelületét (4.15. ábra).



4.15. 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. (4.16. ábra)



4.16. ábra. Adatbányászat eredményei

5. fejezet

Összefoglalás

A dolgozat bemutatta, hogy mi is a Robotic Process Automation, annak elméletét, előnyeit, implementációját és gyakorlati alkalmazását. Felmérésre kerültek gyakran ismételt folyamatok, ezeket a dolgozathoz tartozó szoftver segítségével lehet generálni, szerkeszteni, illetve egy robot segítségével ismételni. Bemutatta a használt eszközöket, fejlesztői környezetet, valamint megindokolta annak kiválasztását.

Ezeket túl nagyvonalakban megismerhettük a folyamatelemzés elméletét és célját, valamint két modelljét; az Alpha-algoritmust és a Heurisztikus Bányászt. A dolgozat ezen részei kifejezetten nehéznek bizonyultak, mind az elméleti része, mind a gyakorlati megvalósítása. A folyamatmodell ábrázolása Petri-hálóként elég komplex műveletsor, talán ez az ami a leginkább időigényes volt.

Sikerült megvalósítani a terveket, illetve olyan funkciókat is, melyek eredetileg nem merültek fel. A továbbiakban tervben van a szoftver továbbfejlesztése és optimalizálása, valamint az egyes részeinek teljeskörű átdolgozása. Ilyenek például a következők.

1. Az adatbányászat eredményeinek megjelenítése szebb formában történjen, kifejezetten a Petri-hálót rajzoló algoritmust kell finomítani.
2. Az időzítő felületet teljesen át kell dolgozni, hogy felhasználóbarátabb legyen.
3. Érdemes egységesíteni a fájlformátumokat, mind a generált folyamatok, mind a kézzel letöltött folyamatok legyenek ugyanolyanok.
4. További folyamatelemzési algoritmusok implementálása.
5. Windows környezethez érdemes lehet service-ként újraírni a szoftver egyes részeit. Ez segítene bizonyos jogosultsági problémák megoldásában.

Összességében a dolgozat eredményeképpen létrejött egy szoftver, melyet használni lehet folyamat-automatizálásra, illetve folyamatelemzésre.

Irodalomjegyzék

- [1] Wil Aalst and Boudewijn Dongen. Discovering petri nets from event logs. 01 2013.
- [2] Kovács Béla. Process mining, 2017. Kutatási jelentés, 44p., EU projekt.
- [3] Embarcadero. Delphi: Ide software overview, 2022. [Online; 2022-Október-17].
- [4] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
- [5] A. Weijters, Wil Aalst, and Alves Medeiros. *Process Mining with the Heuristics Miner-algorithm*, volume 166. 01 2006.
- [6] Gajic Zarko. Delphi history from pascal to embarcadero delphi xe 2, 2021.

Adathordozó használati útmutató

A szakdolgozathoz tartozó adathordozó a következőket tartalmazza:

1. **Szoftver futtatható állománya**

Elérési útvonal: *root:/bin/Szakdolgozat.exe*

2. **Szoftver forráskódja**

Elérési útvonal: *root:/src/*

3. **Szoftverhez tartozó kiegészítő állományok**

Elérési útvonal: *root:/datamining/*

Bemutató elemzések találhatók itt, használt előtt egy mappába ki kell csomagolni őket.

4. **Szakdolgozat dokumentum**

Elérési útvonal: *root:/szakdolgozat.pdf*

5. **Szakdolgozat L^AT_EX forrás**

Elérési útvonal: *root:/doc/*

A szoftver használatához az adathordozóról a számítógépre egy külön könyvtárba szükséges másolni a "Szakdolgozat.exe" fájlt. A folyamatokhoz tartozó bizonyos funkciók (pl. időzítés) csak akkor működnek megfelelően, ha rendszergazdaként futtatjuk, hiszen erre a programra is érvényesek a Windows korlátozásai.