UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Workflow Management

propusă de

Rareș-Alexandru Stan

Sesiunea: *Iulie, 2017*

Coordonator științific

Lect. Dr. Oana Captarencu

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ

Workflow Management

Rareș-Alexandru Stan

Sesiunea: *Iulie, 2017*

Coordonator științific
Lect. Dr. Oana Captarencu

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul „Workflow Management" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imaginile etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,
30 iunie 2017

Absolvent,
Rareș-Alexandru Stan

_____

(semnătura în original)

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „Workflow Management", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza" din Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,
30 iunie 2017

Absolvent,
Rareș-Alexandru Stan

(semnătura în original)

Cuprins

Introducere

Un flux de lucru (sau proces workflow) reprezintă un set ordonat de acțiuni, executate de anumite resurse, folosind și producând date specifice, pentru obținerea unui rezultat final. Analiza fluxurilor de lucru are drept scop verificarea unor proprietăți precum: terminarea corectă a procesului, lipsa blocajelor și posibilitatea de a executa toate acțiunile la un moment dat.

În cadrul acestei lucrări vom prezenta un sistem de administrare a fluxurilor de lucru, "Workflow Management", care va permite modelarea, analizarea și execuția fluxurilor de lucru, într-o manieră cât mai accesibilă și intuitivă, care să nu necesite cunoștințe despre metodele de verificare ale acestora.

Un sistem de administrare a fluxurilor de lucru reprezintă un sistem software ce permite definirea, controlul, execuția și în unele cazuri verificarea fluxurilor de lucru. Pentru a defini un flux de lucru, se pot utiliza metode grafice (grafuri, diagrame UML, limbajul BPMN ([1])), reprezentări textuale (limbaje precum XML, XPDL ([2])) sau limbaje specifice sistemului. Pentru execuția și controlul fluxurilor de lucru, sistemul va indica acțiunile curente, cele care s-au terminat de executat și cele ce pot fi executate la un anumit moment. Există la ora actuală numeroase sisteme comerciale de adminstrare a fluxurilor de lucru (precum Apache Taverna ([3]), Bonita BPM ([4])), dar acestea nu oferă metode de verificare formală a corectitudinii proceselor workflow executate.

Pentru a putea analiza corectitudinea logică a unui flux de lucru este necesară utilizarea unei metode formale. Au fost propuse diverse metode, precum: logici termporale ([5]), algebre de procese ([6, 7]), rețele Petri ([8]). Rețelele Petri, introduse de C.A. Petri în 1962, constituie o metodă formală de modelare și verificare a sistemelor, dispunând și de o reprezentare grafică intuitivă. Diverse clase speciale de rețele Petri au fost propuse pentru modelarea flxurilor de lucru ([9, 8]). În această lucrare vom folosi rețelele Workflow, introduse în ([8])

Sisteme de administrare a fluxului de lucru care utilizează rețele Petri sunt YAWL (Yet Another Workflow Language) și Woflan. YAWL este un sistem în care reprezentarea fluxului de lucru se face în manieră grafică si permite verificarea corectitudinii proceselor. Dezavantajul acestuia este complexitatea procesului de definire a unui flux de lucru. Cel de-al doilea sistem (Woflan) permite verificarea fluxurilor de lucru specificate sub formă de rețele Petri sau poate fi folosit împreună cu o altă aplicație, cu care utilizatorul va modela fluxul de lucru folosind un limbaj specific acesteia. Aplicația Woflan preia definiția fluxului de lucru, îl transformă într-o rețea Petri și îi verifică corectitudinea. Dezavantajul acestei abordări este că implică utilizarea a două aplicații.

Sistemul software propus în cadrul acestei lucrări oferă posibilitatea de a crea fluxuri de lucru în mod grafic: acțiunile și structurile de control sunt reprezentate prin noduri de diverse forme, iar legătura dintre acțiuni este reprezentată prin arce direcționate, interfața fiind disponibilă într-un browser web. Datele necesare executării unei acțiuni și cele ce vor fi returnate după execuție sunt specificate la nivelul fiecărui nod. Specificarea resurselor ce vor executa o acțiune se face la nivelul fiecărui nod, prin indicarea rolului și eventual prin enumerarea angajaților pe care i-ar prefera utilizatorul. Pentru verificarea corectitudinii, reprezentarea grafică a fluxului de lucru este transformată într-o rețea Petri, iar pentru aceasta se aplică algoritmul de verificare. Daca fluxul de lucru este corect, atunci el poate fi pus în execuție, iar aplicația se va ocupa de oferirea și cererea de date de intrare și de ieșire, pentru fiecare acțiune și va urmări executarea în ordine a acțiunilor. Când o acțiune este terminată, administratorul fluxului verifică datele obținute și alege noua acțiune care va fi executată, dintr-o listă de acțiuni posibile. Fiecare angajat are acces la o listă cu toate acțiunile care i-au fost asignate de către sistem și posibilitatea de a interacționa cu ele, pe lângă încărcarea și descărcarea datelor necesare.

Față de Woflan, această aplicație oferă în plus posibilitatea definirii și verificării fluxurilor de lucru, specificarea resurselor și a datelor de intrare și de ieșire ale fiecărei acțiuni, oferă posibilitatea de punere în execuție a fluxului de lucru și administrarea acțiunilor asignate fiecărui angajat și nu necesită instalarea unei aplicații pentru fiecare utilizator, interfața fiind disponibilă într-un browser web. Spre deosebire de YAWL, oferă o interfață mai ușor de folosit și o modalitate mai simplă de definire a unui flux de lucru.

Aplicația este împărțită în aplicație web client și aplicație server. La implementarea părții client am folosit AngularJS1.x pentru modularitatea codului și arhitecturii MV*, AngularMaterial oferind posibilitatea unui design adaptiv la diferite dimensiuni de ecran, iar pentru vizualizarea și interacțiunea cu fluxul de lucru am folosit biblioteca visjs. Implementarea serverului este făcută în python 3.x și am folosit modulul flask, impreună cu două extensii ale acestuia. Cu ajutorul extensiei flask-RESTful am putut crea foarte ușor un server REST, fiecărui URL îi corespunde o clasă ce are implementate metode cu numele operațiilor http (get, put, post, delete, etc.). Pentru baza de date am folosit postgresql, iar pentru comunicarea între server și baza de date am folosit ORM-ul sqlalchemy, împreună cu extensia pentru flask, flask-sqlalchemy.

În Capitolul 1 vom prezenta principalele noțiuni teoretice referitoare la rețele Petri și rețele Workflow. Capitolul 2 prezintă aplicația "Workflow Management", funcționalitățile acesteia, arhitectura și diverse detalii de implementare. În final vom prezenta concluziile acestei lucrări.

1 Fluxuri de lucru: modelare utilizând rețele Petri

În acest capitol vom defini noțiunea de flux de lucru (sau proces workflow), utilitatea acestora, precum și o modalitate de reprezentare a fluxurilor de lucru. În subcapitolul 1.2 vom defini rețelele Petri împreună cu câteva proprietăți și teoreme. În subcapitolul 1.3 vom defini rețelele Workflow, o clasă specială de rețele Petri utilizată pentru modelarea și analiza fluxurilor de lucru. Pentru a descrie execuția corectă a unui flux de lucru va fi introdusă proprietatea de corectitudine a rețelelor Workflow și apoi vor fi prezentate metode de verificare a corectitudinii.

1.1 Fluxuri de lucru. Sisteme de administrare a fluxurilor de lucru

Un flux de lucru reprezintă automatizarea unui set ordonat de acțiuni, executate de anumite resurse, folosind anumite date specifice, pentru obținerea unui rezultat final.

Fiecare flux de lucru se execută pentru un caz diferit, unde un caz reprezintă subiectul operațiilor din fluxul de lucru.

Cea mai mică unitate logică dintr-un flux de lucru se numește *acțiune*. Dependențele dintre acțiuni determină ordinea acestora de execuție. Acestea sunt executate de către *resurse*, ce pot fi de mai multe tipuri: umane, software, hardware, etc. O *acțiune* ce corespunde unui anumit caz se numeste *unitate de lucru*, iar dacă acesteia îi este atribuită o resursă, se numește *activitate*.

Structurile de control al execuției descriu dependența logică între acțiuni. Cele mai utilizate structuri de control (reprezentate grafic în Figura 1) sunt: secvența, AND-split, AND-join, OR-split și OR-join.

- **Secvența**: reprezintă o dependență secvențială intre două acțiuni. Dacă reprezentăm grafic fiecare nod sub forma unui dreptunghi, dependența secvențială se va reprezenta grafic sub forma unui arc între cele două. În Figura 1.b se specifică faptul că acțiunea B trebuie să se producă după ce se termină acțiunea A.

- **AND-split**: reprezintă execuția în paralel a următoarelor acțiuni. Se va reprezenta grafic print-o elipsă ce conține textul "AND-split". În Figura 1.c se specifică faptul că acțiunile $\alpha_1, \alpha_2, \ldots, \alpha_n$ își vor începe execuția simultan, dupa ce se termină acțiunea A.

- **AND-join**: reprezintă așteptarea terminării mai multor acțiuni și apoi inceperea unei singure acțiuni. Se va reprezenta grafic printr-o elipsă ce conține textul

"AND-join". În Figura 1.d se specifică faptul că se va aștepta sfârșitul acțiunilor $\beta_1, \beta_2, \ldots, \beta_m$ și apoi se va produce acțiunea A.

- **OR-split**: reprezintă producerea uneia dintre acțiunile următoare, utilizatorul alegând acțiunea următoare. Se va reprezenta grafic printr-o elipsă ce conține textul "OR-split". În Figura 1.e se specifică faptul că după terminarea acțiunii A se va produce una din acțiunile $\sigma_1, \sigma_2, \ldots, \sigma_k$.

- **OR-join**: reprezintă execuția acțiunii următoare, indiferent de ce acțiune și-a terminat execuția, din înaintea acestui nod. Se va reprezenta grafic printr-o elipsă ce conține textul "OR-join". În Figura 1.f se specifică faptul că după terminarea uneia dintre acțiunile $\gamma_1, \gamma_2, \ldots, \gamma_l$ se va produce acțiunea A.

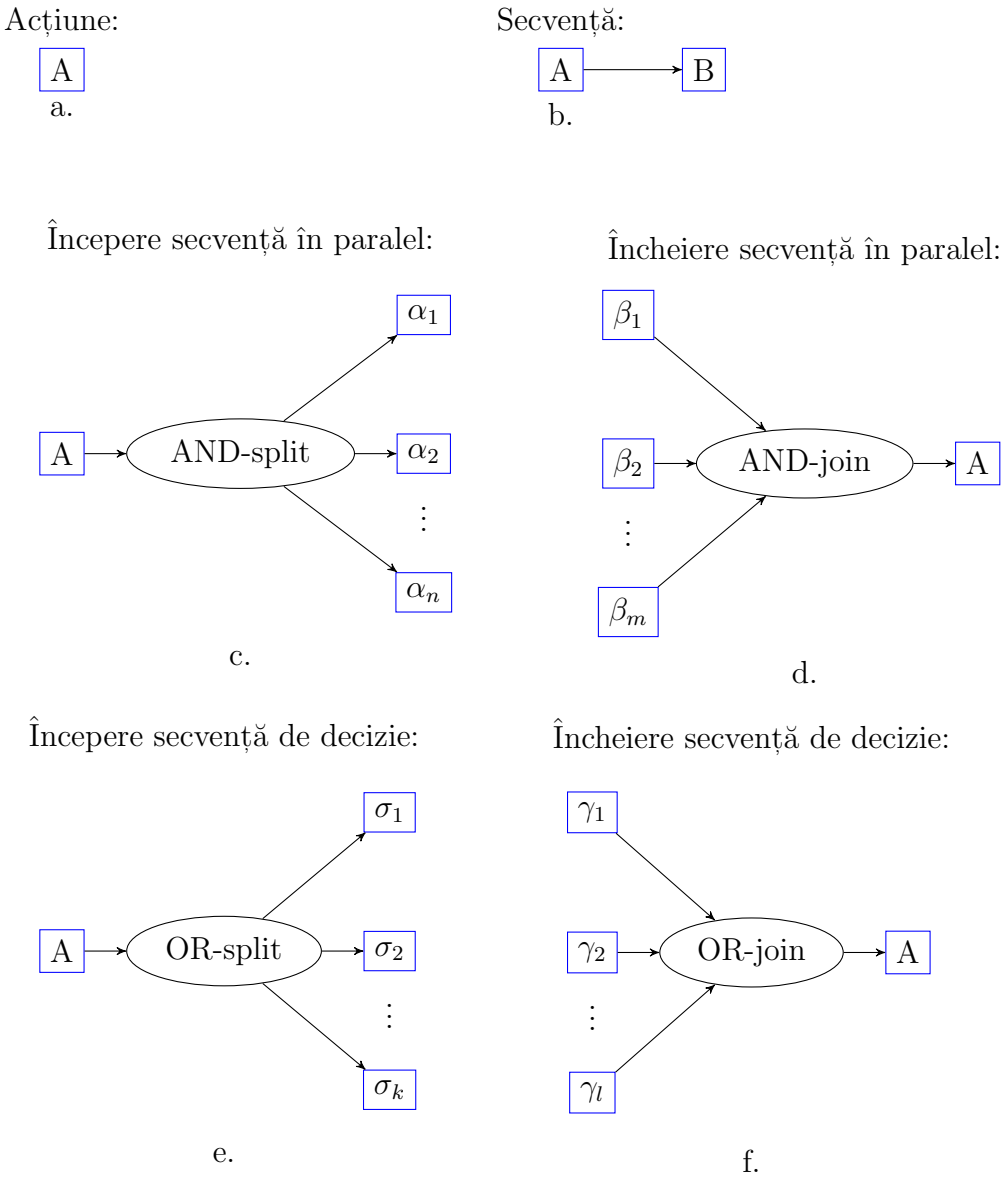

Figura 1: O reprezentare grafică a structurilor de control ale unui flux de lucru

Un flux de lucru este structurat în mai multe perspective:

- *Perspectiva proces* descrie acțiunile și ordinea lor de execuție. Reprezentarea grafică din Figura 1 oferă o bună vizualizare a acțiunilor și dependențelor dintre acestea.
- *Perspectivei resurselor* descrie resursele, modul de organizare a acestora și modul lor de alocare pentru execuția acțiunilor.
- *Perspectiva datelor* oferă o vizualizare a datelor folosite pentru controlul execuției, precum și datele create sau utilizate de către resurse pentru indeplinirea acțiunilor.

Verificarea unui flux de lucru are drept scop obținerea unor proprietăți calitative, înainte de punerea în execuție sau începerea implementării acestuia, cum ar fi existența blocajelor, dacă se poate incheia cu succes un caz, odată ce a fost pus in execuție, dacă se pot executa toate acțiunile definite în proces, etc.

Un sistem de administrare a fluxurilor de lucru (WfMS) este un sistem software ce se ocupă cu definirea, controlul și execuția acestora. Sistemul trebuie să asigure distribuția unităților de lucru către resursele potrivite pentru execuția lor eficientă, în ordinea specificată. Astfel de WfMS-uri sunt YAWL și Woflan, care oferă o reprezentare grafică și modalități de verificare a fluxurilor de lucru.

1.2 Rețele Petri

Rețelele Petri reprezintă o metodă formală folosită pentru modelarea și verificarea sistemelor concurente/distribuite. Prin sistem înțelegem o combinație de componente ce acționează împreună pentru a efectua o funcție ce nu ar fi posibilă în absența uneia din părțile individuale.

Definiție 1. O rețea Petri este un 4-uplu $N = (P, T, F, W)$ astfel încât:

1. P este mulțimea de locații, T este mulțimea de tranziții, $P \cap T = \emptyset$;
2. $F \subseteq (P * T) \cup (T * P)$ relația de flux (mulțimea arcelor);
3. $W : (P * T) \cup (T * P) \to \mathbb{N}$ ponderea arcelor ($W(x, y) = 0$ ddacă $(x, y) \notin F$)

Definiție 2. Dacă $x \in P \cap T$ atunci:

- Premulțimea lui x (sau mulțimea elementelor input pentru x): $\bullet x = \{y | (y, x) \in F\}$;
- Postmulțimea lui x (sau mulțimea elementelor output pentru x): $x\bullet = \{y | (x, y) \in F\}$;

Definiție 3. O rețea Petri este pură dacă $\forall x \in P \cup T, \bullet x \cap x\bullet = \emptyset$.

Definiție 4. (Marcare, rețele marcate)

- Fie $N = (P, T, F, W)$ o rețea Petri. O marcare a lui N este o funcție $M : P \to \mathbb{N}$.
- Fie $N = (P, T, F, W)$ o rețea Petri și $M_0 : P \to \mathbb{N}$. Atunci (N, M_0) se numește rețea Petri marcată.

Tranzițiile reprezintă acțiunile sau evenimentele din sistemul modelat. Locațiile pot fi de tip intrare sau de tip ieșire. Cele de tip intrare reprezintă parametri, resurse necesare unei acțiuni, pe când locațiile de tip ieșire sunt datele, parametri generați dupa executarea unei acțiuni. Ponderea arcelor reprezintă numarul necesar de resurse de un anumit tip (arce de tip intrare) sau numarul de resurde de un anumit tip generate (arce de tip ieșire). Valorile funcției de marcare, pentru o locație pot indica resurse sau valori booleene.

Definiție 5. Fie $N = (P, T, F, W)$ o rețea Petri, M o marcare a lui N și $t \in T$ o tranziție a lui N.

- Tranziția t este posibilă la marcarea $M (M [t\rangle_N)$ dacă $W(p, t) \leq M(p), \forall p \in \bullet t$
- Dacă t este posibilă la marcarea M, atunci t se poate produce, rezultând o nouă marcare $M' (M [t\rangle_N M')$, unde $M'(p) = M(p) = W(p, t) + W(t, p), \forall p \in P$

Fie secvența $u \in T^*, t \in T$ și marcarea M.

- secvența vidă de tranziții ϵ este secvență de tranziții posibilă la M și $M [\epsilon\rangle M$;
- dacă u este secvență de tranziții posibilă la $M, M [u\rangle M'$ și $M' [t\rangle M''$, atunci ut este secvență de tranziții posibilă la M și $M [ut\rangle M''$.

Dacă $\exists \sigma \in T^*$ astfel încât $M [\sigma\rangle M'$, se mai notează $M [*\rangle M'$.

Fie $\gamma = (N, M_0)$ o rețea Petri marcată. Se definesc următoarele funcții:

- $t^- : P \to \mathbb{N}, t^-(p) = W(p, t), \forall p \in P, t \in T$
- $t^+ : P \to \mathbb{N}, t^+(p) = W(t, p), \forall p \in P, t \in T$
- $\triangle t : P \to \mathbb{Z}, \triangle t(p) = W(t, p) - W(p, t), \forall p \in P, t \in T$

Dacă $\sigma \in T^*$ este o secvență de tranziții, se definește $\triangle \sigma : P \to \mathbb{Z}$:

- Dacă $\sigma = \epsilon$, atunci $\triangle \sigma$ este funcția identic 0.
- Dacă $\sigma = t_1, \cdots, t_n$, atunci $\triangle \sigma = \sum_{i=1}^{n} \triangle t_i, t \in T$.

Definiție 6. Fie $\gamma = (N, M_0)$ o rețea Petri marcată. O marcare M' este accesibilă din marcarea M, dacă există o secvență finită de apariție σ astfel încât $M [\sigma\rangle M'$.

Mulțimea marcărilor accesibile dintr-o marcare M, în γ, se notează $[M\rangle_\gamma$

Definiție 7. Marcarea M este accesibilă în γ, dacă M este accesibilă din marcarea inițială M_0.

Mulțimea marcărilor accesibile în γ se notează $[M_0\rangle_\gamma$

Propoziție 1. Fie M, M' și L marcări, $\sigma \in T^*$ o secvență de tranziții, posibilă la M.

- Dacă σ finită și $M\,[\sigma\rangle\,M'$, atunci $(M+L)\,[\,\sigma\rangle(M'+L)$.
- Daca σ infinită și $M\,[\sigma\rangle$, atunci $(M+L)\,[\sigma\rangle$.

Definiție 8. Fie M, M' două marcări.

- $M \geq M' \Leftrightarrow M(p) \geq M'(p) \forall p \in P$.
- $M > M' \Leftrightarrow M \geq M'$ și $\exists p \in P : M(p) > M'(p)$.

Propoziție 2. Fie M, M' două marcări astfel încât $M' \geq M$, atunci $\forall \sigma \in T^*$ secvență de tranziții posibilă la marcarea M este posibilă și la marcarea M'.

Definiție 9. Fie $\gamma = (N, M_0)$ o rețea Petri marcată.

- O locație p este mărginită dacă: $\exists n \in \mathbb{N}$ astfel încât $M(p) \leq n, \forall M \in [M_0\rangle$
- Rețeaua marcată γ este mărginită daca $\forall p \in P, p$ este mărginită.

Propoziție 3. O rețea Petri marcată $\gamma = (N, M_0)$ este mărginită ddacă mulțimea $[\,M_0\rangle$ este finită.

Propoziție 4. Dacă $\gamma = (N, M_0)$ este mărginită, $\nexists M_1, M_2 \in [M_0\rangle$ astfel încât $M_1\,[*\rangle\,M_2$ și $M_2 : M_1$.

Definiție 10. (pseudo-viabilitatea) Fie $\gamma = (N, M_0)$ o rețea Petri marcată.

- O tranziție $t \in T$ este pseudo-viabilă din marcarea M, daca $\exists M' \in [M\rangle$ astfel încât $M'\,[t\rangle$.
- O tranziție $t \in T$ este pseudo-viabilă daca este pseudo-viabilă din $M_0 \Leftrightarrow \exists M \in [M_0\rangle$ astfel încât $M\,[t\rangle$. O tranziție care nu este pseudo-viabilă se numește moartă.
- Rețeaua marcată γ este pseudo-viabilă daca $\forall t \in T, t$ este pseudo-viabilă.

Definiție 11. (blocaje) Fie $\gamma = (N, M_0)$ o rețea Petri marcată.

- O marcare M a rețelei γ este moartă daca $\nexists t \in T$ astfel încât $M\,[\,t\rangle$.

- Rețeaua γ este fără blocaje, dacă nu există marcări accesibile moarte.

Definiție 12. (viabilitate) Fie $N = (P, T, F, W)$ o rețea Petri și $\gamma = (N, M_0)$ o rețea Petri marcată.

- O tranziție $t \in T$ este viabilă daca $\forall M \in [M_0\rangle, t$ este pseudo-viabilă din M ($\exists M' \in [M\rangle$ astfel încât $M'[t\rangle$).
- Rețeaua marcată γ este viabilă dacă $\forall tinT, t$ este viabilă.

Definiție 13. (marcare acasă) Fie $\gamma = (N, M_0)$ o rețea marcată și H marcare a sa. H este marcare acasă dacă $\forall M \in [M_0\rangle, H \in [M\rangle$

Definiție 14. (reversibilitate) Rețeaua marcată γ este reversibilă dacă marcarea sa inițială este marcare acasă.

Propoziție 5. O rețea este reversibilă $\Leftrightarrow \forall M \in [M_0\rangle, M$ este marcare acasă.

Fie $\gamma = (N, M_0)$ o rețea Petri marcată.

Propoziție 6. Orice rețea marcată viabilă este și pseudo-viabilă.

Propoziție 7. Orice rețea marcată viabilă, având cel puțin o tranziție, este fără blocaje.

Propoziție 8. Daca o rețea fără locații izolate este viabilă, atunci orice locație poate fi marcată, din orice marcare accesibilă.

Propoziție 9. O rețea marcată reversibilă este viabilă ddacă este pseudo-viabilă.

Propoziție 10. O rețea marcată reversibilă este fără blocaje.

Definiție 15. Fie $\gamma = (N, M_0)$ o rețea Petri marcată. Tranzițiile t_1, t_2 se produc în secvență dacă $M_1[t_1\rangle, \neg M_1[t_2\rangle$ și $M_1[t_1\rangle M_2[t_2\rangle M_3$.

Definiție 16. Fie $\gamma = (N, M_0)$ o rețea Petri marcată. O mulțime de tranziții $U \subset T$ este concurent posibilă la o marcare M a lui N dacă $\sum_{t \in U} W(p, t) \leq M(p), \forall p \in P$.

Propoziție 11. Dacă $U \subset T$ este o mulțime de tranziții concurent posibile la o marcare M, orice permutare σ a tranzițiilor din U este o secvență de apariție posibilă la M ($M[\sigma\rangle M'$).

Propoziție 12. Fie N o rețea pură, $t_1, t_2 \in T$ și M o marcare a lui N. Atunci t_1 și t_2 sunt concurent posibile la marcarea M dacă și numai dacă $M[t_1 t_2\rangle M'$ și $M[t_2 t_1\rangle M'$.

Cu ajutorul structurilor de acoperire, cum ar fi graful de accesibilitate sau arborele de acoperire, putem determina mai usor anumite proprietăți ala rețelelor Petri, cum ar fi mărginirea rețelei, pseudo-viabilitatea, etc.

Definiție 17. (graf de accesibilitate) Fie $\gamma = (N, M_0)$ o rețea Petri marcată. Graful de accesibilitate pentru γ este un graf orientat cu arce etichetate, $\mathcal{GA} = (V, E, l_V)$, astfel încât:

- $V = [M_0\rangle$
- $E = \{(M, M') | \exists t \in T, M\,[t\rangle\, M'\}$
- $l_E : E \to T, \forall (M, M') \in E, l_E(M, M') = t$, dacă $M\,[t\rangle\, M'$

Notăm un arc etichetat $l_E(M, M') = t, \forall (M, M') \in E, \forall t \in T$ cu (M, t, M').

Propoziție 13. O rețea Petri marcată $\gamma = (N, M_0)$ este mărginită dacă și numai dacă graful său de accesibilitate $\mathcal{GA}(\gamma)$ are un număr finit de noduri.

Propoziție 14. O rețea Petri marcată $\gamma = (N, M_0)$ este fără blocaje dacă și numai dacă graful său de accesibilitate $\mathcal{GA}(\gamma)$ nu conține noduri fără succesori.

Propoziție 15. O rețea Petri marcată $\gamma = (N, M_0)$ este pseudo-viabilă dacă și numai dacă graful său de accesibilitate $\mathcal{GA}(\gamma)$ conține arce etichetate cu toate tranzițiile rețelei.

Propoziție 16. O rețea Petri marcată $\gamma = (N, M_0)$ este viabilă dacă și numai dacă în graful său de accesibilitate $\mathcal{GA}(\gamma) = (V, E, l_E), \forall M \in V, \exists M(M, t_1, M_1)$ $M_1(M_1, t_2, M_2)M_2, \ldots, M_{k-1}(M_{k-1}, t_{k-1}, M_k)M_k$ drum, astfel încât secvența $t_1, t_2, \ldots, t_{k-1}, t_k$ conține toate tranzițiile din γ.

Propoziție 17. O rețea Petri marcată $\gamma = (N, M_0)$ este reversibilă dacă și numai dacă graful său de accesibilitate $\mathcal{GA}(\gamma)$ este tare conex.

Pentru a descrie comportamentul rețelelor cu număr infinit de marcări se introduc arborii și grafurile de acoperire.

Definiție 18. Fie N o rețea Petri, M_1 și M_2 două marcări ale sale. M_2 acoperă M_1 dacă $M_2 \geq M_1$

Definiție 19. O rețea Petri marcată $\gamma = (N, M_0)$ și M o marcare a lui N. M este acoperibilă în γ dacă $\exists M' \in [M_0\rangle$ care acoperă pe M.

Vom introduce un simbol pentru infinit (ω) și avem: $\overline{\mathbb{N}} = \mathbb{N} \cup \omega$. Au loc următoarele proprietăți:

- $\omega + n = \omega - n = \omega, \forall n \in \mathbb{N}$
- $\omega + \omega = \omega - \omega = \omega$
- $\omega * n = n * \omega = \omega, \forall n \in \mathbb{N}, n > 0$

- $\omega * 0 = 0$
- $\omega > n, \forall n \in \mathbb{N}$

Fie $\overline{\mathbb{N}}^P = \{M | M : P \to \overline{\mathbb{N}}\}$.

Fie $\tau = (V, E)$ un arbore.

- Dacă $v \in V$ este un nod, v^+ este mulțimea succesorilor lui v ($v^+ = \{v' \in V | \exists (v, v') \in E\}$).
- $d(v, v')$ este drumul (unic) de la nodul v la nodul v'.

Definiție 20. (arbore de acoperire) Fie $\gamma = (N, M_0)$ o rețea Petri marcată. Se numește arbore de acoperire al rețelei γ orice arbore $\tau_\gamma = (V, E, l_V, l_E)$, ce satisface următoarele proprietăți:

1. $l_V : V \to \overline{\mathbb{N}}^P, l_E : E \to T$;
2. rădăcina v_0 a lui $\tau_\gamma \gamma$ este etichetată cu $M_0 : l_V(v_0) = M_0$;
3. pentru orice nod v cu eticheta M ($l_V(v) = M$) are loc:
   (a) $|v^+| = 0$ (v este nod frunză), dacă $\nexists t \in T$ astfel încât $M[t\rangle$ sau $\exists v' \in d(v_0, v), v \neq v'$ cu eticheta M' astfel încât $M = M'$;
   (b) $|v^+| = \{t \in T | M[t\rangle\}$, altfel;
4. pentru orice $v \in V$ cu $|v^+| > 0, l_V(v) = M$ și $\forall t \in T : M[t\rangle$ există $v' \in V, l_V(v') = M'$ astfel încât:
   (a) $(v, v') \in E$;
   (b) $l_E(v, v') = t$;
   (c) Fie $\overline{M} = M + \triangle(t)$. $\forall p \in P$ are loc:
      - $M'(p) = \omega$, dacă $\exists v'' \in d_\tau(v_0, v)$ astfel încât $l_V(v'') = M'', M'' \leq \overline{M}$ și $M''(p) < \overline{M}(p)$;
      - $M'(p) = \overline{M}(p)$, altfel

Fie $\gamma = (N, M_0)$ o rețea Petri marcată și $\tau_\gamma = (V, E, l_V, l_E)$ arborele ei de acoperire.

- Dacă $(v_1, v_2) \in E, l_E(v_1, v_2) = t, l_V(v_1) = M_1$ și $l_V(v_2) = M_2$, atunci vom nota $v_1 : M_1 \xrightarrow{t} v_2 : M_2$.
- Fie $v \in V$ cu $l_V(v) = M$. $\Omega(M) = \{p \in P | M(p) = \omega\}$.

- $Lab(\tau_\gamma)$ este mulțimea etichetelor nodurilor arborelui de acoperire corespunzător lui $\gamma : Lab(\tau_\gamma) = \{l_V(v) | v \in V\}$

Propoziție 18. Fie $\gamma = (N, M_0)$ o rețea Petri marcată și $\tau_\gamma = (V, E, l_V, l_E)$ arborele său de acoperire. Au loc următoarele proprietăți:

1. $\tau_\gamma\gamma$ este finit ramnificat
2. Fie $v_{i_0}, v_{i_1}, \ldots, v_{i_m}$ noduri distincte două câte două astfel încât $v_{i_j} \in d(v_0, v_{i_{j+1}})$ pentru orice $0 \leq j \leq m-1$
   - Dacă $l_V(v_{i_0}) = l_V(v_{i_1}) = \cdots = l_V(v_{i_m})$, atunci $m \leq 1$;
   - Dacă $l_V(v_{i_0}) < l_V(v_{i_1}) < \cdots < l_V(v_{i_m})$, atunci $m \leq |P|$;
3. $\tau(\gamma)$ este finit

Lema 1. Fie $\gamma = (N, M_0)$ o rețea Petri marcată și $\tau_\gamma = (V, E, l_V, l_E)$ arborele său de acoperire. Dacă $v_1, v_2 \in V, l_V(v_1) = M_1, l_V(v_2) = M_2, w \in T^*$ și $v_1 : M_1 \xrightarrow{w} v_2 : M_2$, atunci $M_2(p) = (M_1 + \triangle w)(p), \forall p \in P \setminus \Omega(M_2)$.

Definiție 21. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, τ_γ arborele său de acoperire și M o marcare. M este acoperibilă în τ_γ dacă $\exists v \in V : l_V(v) \geq M$.

Lema 2. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, $\tau_\gamma = (V, E, l_V, l_E)$ arborele său de acoperire. Orice marcare accesibilă în γ este acoperibilă în τ_γ.

Lema 3. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, $\tau_\gamma = (V, E, l_V, l_E)$ arborele său de acoperire și M o marcare a lui N. Dacă M este acoperibilă în τ_γ, atunci M este acoperibilă în γ.

Teoremă 1. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, $\tau_\gamma = (V, E, l_V, l_E)$ arborele ei de acoperire și M o marcare a lui N. M este acoperibilă în τ_γ dacă și numai dacă M este acoperibilă în γ.

Propoziție 19. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, $\tau_\gamma = (V, E, l_V, l_E)$ arborele ei de acoperire și $t \in T$. Atunci are loc: $(\exists M \in [M_0\rangle : M\,[t\rangle) \Leftrightarrow (\exists (v, v') \in E : l_E(v, v') = t)$.

Cu ajutorul arborelui de acoperire $\tau_\gamma = (V, E, l_V, l_E)$ putem verifica cu mare ușurință dacă o rețea Petri $\gamma = (N, M_0)$ îndeplinește următoarele proprietăți:

- **Mărginirea unei locații**: O locație $p \in P$ este mărginită dacă și numai dacă $\forall v \in V, l_V(v)(p) \neq \omega$
- **Mărginirea rețelei**: Rețeaua γ este mărginită dacă și numai dacă $Lab(\tau_\gamma) \subseteq \mathbb{N}^P$

- **Pseudo-viabilitatea unei tranziții**: O tranziție t a lui γ este pseudo-viabilă dacă și numai dacă $\exists(v, v') \in E : l_E(v, v') = t$

- **Acoperibilitatea unei marcări**: O marcare M a lui γ este acoperibilă dacă și numai dacă $\exists v \in V : l_V(v) = M'$ și $M' \geq M$

1.3 Modelarea fluxurilor de lucru utilizând rețele Workflow

Pentru determinarea unor proprietăți calitative ale fluxului de lucru, cum ar fi existența blocajelor, încheierea execuției cu succes a unui caz, posibilitatea de a se executa toate acțiunile din proces, a fost propusă modelarea perspectivei proces folosind rețele Petri și corelarea proprietăților fluxurilor de lucru cu cele ale rețelelor Petri. Astfel au fost introduse rețelele Workflow ([8]), în care o acțiune din fluxul de lucru este modelată printr-o tranziție, un caz printr-un punct în rețea, precondițiile și postcondițiile prin locații, unitățile de lucru prin tranziții posibile intr-o anumită stare, activitatea prin tranziție ce se execută, iar structurile de control ale execuției prin locații sau tranziții.

Definiție 1. O rețea Workflow (WF-rețea) este o rețea Petri $PN = (P, T, F)$ astfel încât:

- P conține o locație input i și o locație output o astfel încât $\bullet i = \emptyset$ și $o\bullet = \emptyset$.

- $\forall n \in P \cup T$, există un drum în PN de la i la n și un drum de la n la o.

Observații: $W(x, y) = 1, \forall(x, y) \in F$
Notații:

- Marcarea inițială, M_0, a unei rețele Workflow: $M_0(i) = 1, M_0(p) = 0, \forall p \in P, p \neq i$. Se notează $M_0 = i$.

- Marcarea finală, M_f, a unei rețele Workflow: $M_f(o) = 1, M_f(p) = 0, \forall p \in P, p \neq o$. Se notează $M_f = o$.

Transformarea celor mai utilizate structuri de control al execuției fluxurilor de lucru, secvența, AND-split, AND-join, OR-split, OR-join, în rețele Workflow (reprezentare grafică în Figura 2):

- **Secvența**: este reprezentată în rețelele Workflow cu ajutorul unei locații, având reprezentare grafică în Figura 2.b.

- **AND-split**: este reprezentat în principal de o tranziție auxiliară, ce leagă tranziția A de următoarele n tranziții, pentru ca următoarele acțiuni să fie simultan puse în execuție, având reprezentarea grafică din Figura 2.c.

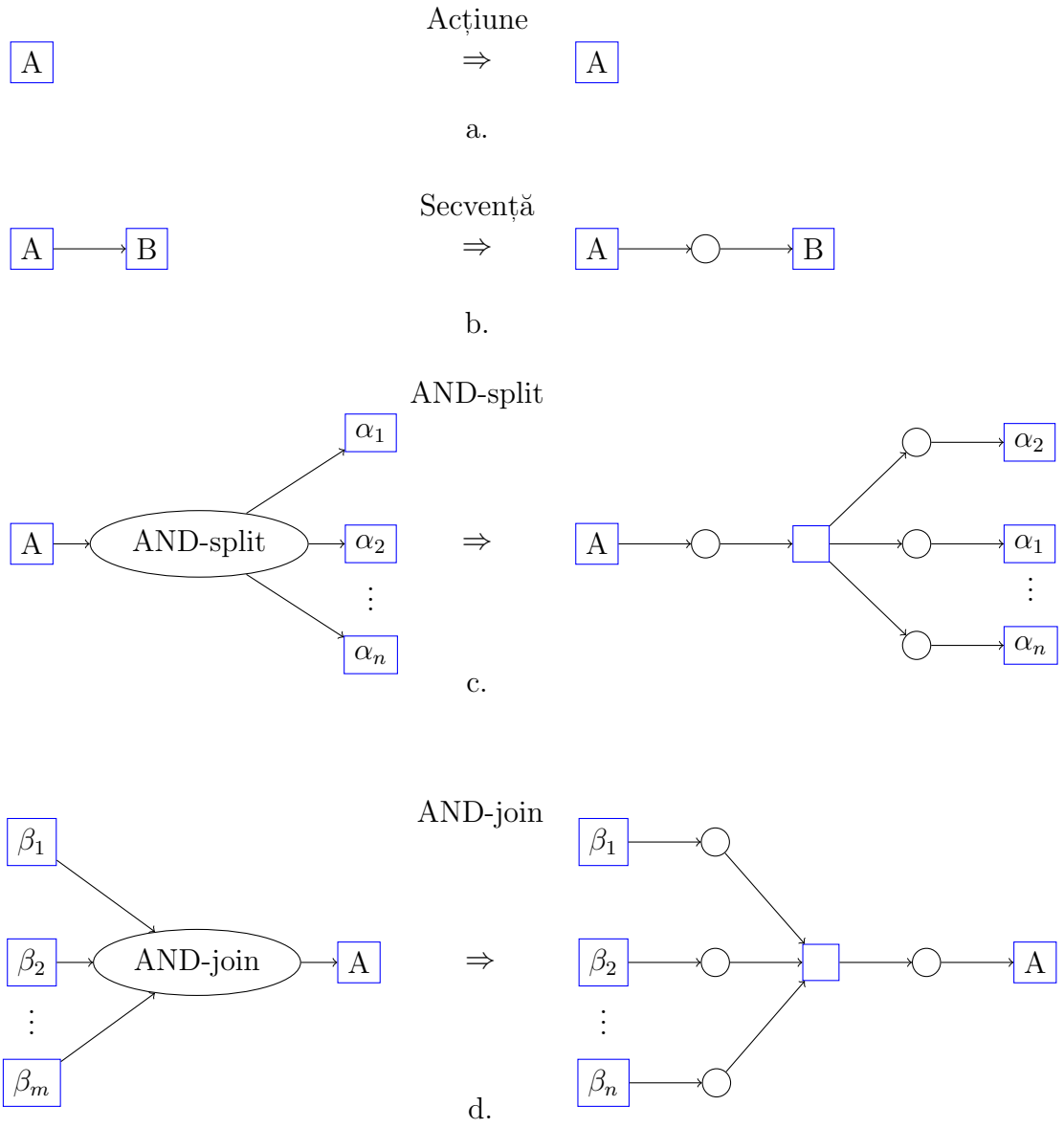

Figura 2: Transformarea structurilor de control în rețele Workflow

- **AND-join**: este reprezentat asemănator cu AND-split, având o tranziție ce leagă cele m tranziții anterioare de tranziția A, pentru a putea aștepta terminarea tuturor tranzițiilor anterioare, având reprezentare grafică în Figura 2.d.

- **OR-split**: are ca și reprezentare o locație urmată de k tranzișii auxiliare, ce sunt apoi legate de acțiunile $\sigma_1, \sigma_2, \ldots, \sigma_k$. Prin utilizarea locației, se va putea executa doar o singură tranziție, din cele ce o urmează, fiind reprezentată grafic în Figura 2.e.

- **OR-join**: este asemănător reprezentării OR-split, doar că cele l acțiuni precedente intră fiecare în câte o locație și spoi acestea intră într-o singură locație, astfel la terminarea uneia din cele l acțiuni, se va pune în execuție acțiunea A. Această

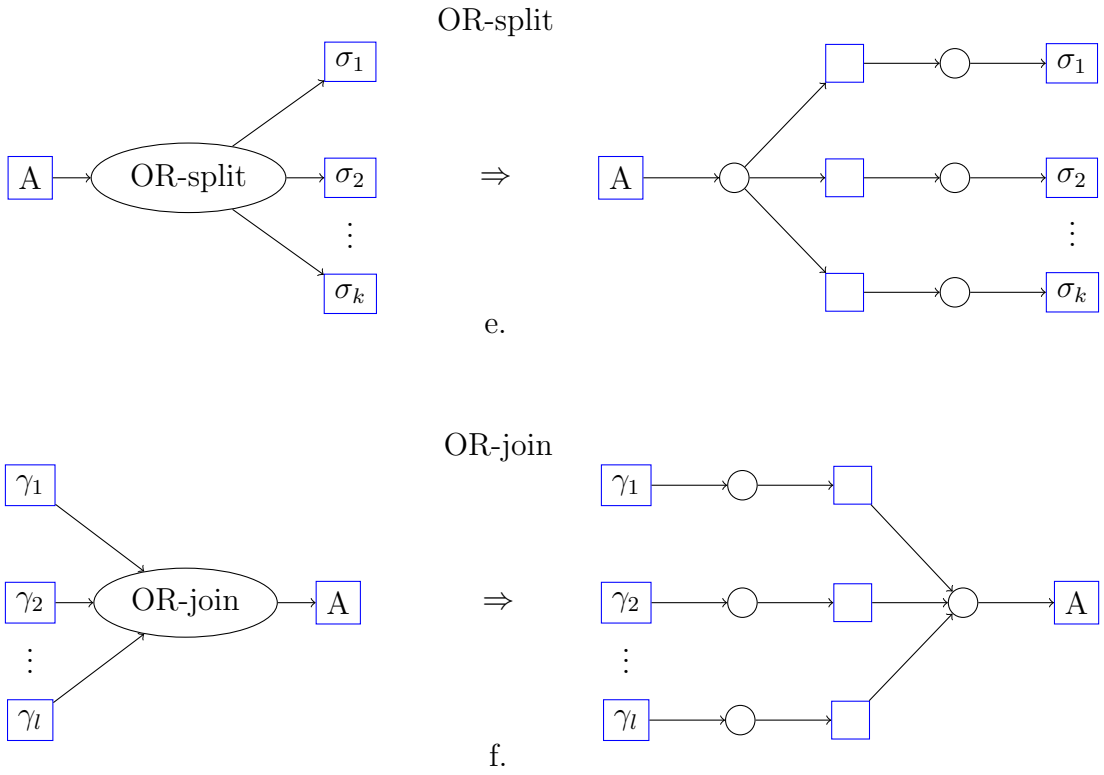

Figura 2: Transformarea structurilor de control în rețele Workflow

reprezentare este ilustrată grafic în Figura 2.f.

Corectitudinea logică a unui flux de lucru se exprimă prin intermediul proprietății de corectitudine a rețelelor Workflow:

- Într-un flux de lucru execuția unui caz trebuie să se poată termina întotdeauna;
- Nu există acțiuni inutile (orice acțiune trebuie să se poată produce la un moment dat).

Definiție 2. O rețea Workflow $PN = (P, T, F)$ este corectă dacă și numai dacă:

- $\forall M \in [i\rangle\, , o \in [M\rangle$ (condiția de terminare corectă)
- $\forall t \in T, t$ este pseudo-viabilă

Definiție 3. Fie $PN = (P, T, F)$ o rețea Workflow. Închiderea rețelei PN este o rețea $\overline{PN} = (\overline{P}, \overline{T}, \overline{F})$, astfel încât:

- $\overline{P} = P$
- $\overline{T} = T \cup \{t^*\}$
- $\overline{F} = F \cup \{(o, t^*), (t^*, i)\}$

Lema 1. Fie $PN = (P, T, F)$ o rețea Workflow pentru care are loc condiția de terminare corectă. Atunci au loc:

- $\forall M \in [i\rangle\, , M \geq o \Rightarrow M = o$
- (PN, i) este mărginită
- mulțimea marcărilor accesibile din (PN, i) coincide cu mulțimea marcărilor accesibile din $(\overline{PN}, i)$
- (PN, i) este pseudo-viabilă dacă și numai dacă $(\overline{PN}, i)$ este pseudo-viabilă

Lema 2. Fie $PN = (P, T, F)$ o rețea Workflow corectă. Atunci $(\overline{PN}, i)$ este o rețea viabilă și mărginită.

Lema 3. Fie $PN = (P, T, F)$ o rețea Workflow. Dacă $(\overline{PN}, i)$ este o rețea viabilă și mărginită, atunci PN este corectă.

Teoremă 1. O rețea Workflow PN este corectă dacă și numai dacă $(\overline{PN}, i)$ este viabilă și mărginită.

Observație: O rețea Workflow este corectă dacă și numai dacă:

- marcarea o este marcare acasă
- rețeaua este pseudo-viabilă

În continuare vom prezenta câteva noțiuni și notații referitoare la grafuri, pe care le vom aplica ulterior pentru graful de accesibilitate al unei rețele Petri.

Fie $\mathcal{G} = (V, A)$ un graf orientat:

- Dacă $a = (v_1, v_2) \in A, s(a) = v_1, d(a) = v_2$
- $\mathcal{DF}$ este o mulșime de drumuri finite în graful $\mathcal{G}$
- $v_1, v_2 \in V, DF(v_1, v_2)$ este mulțimea de drumuri finite ce leagă nodurile v_1 și v_2 în $\mathcal{G}$

Fie $\mathcal{G} = (V, A)$ un graf orientat. O componentă tare conexă a grafului $\mathcal{G}$ este un subgraf $\mathcal{G}^*$ indus de mulțimea de noduri $V^* \subseteq V$, astfel încât:

- V^* este o mulțime de noduri tari conexe $(\forall v_1, v_2 \in V^* : DF(v_1, v_2) \neq \emptyset)$
- $\forall V' \subseteq V$, V' tare conex și $V^* \subseteq V' \Rightarrow V^* = V'$

O componentă tare conexă c este o componentă tare conexă terminală dacă și numai dacă $\forall v \in c, \forall a \in A, s(a) = v \Rightarrow d(a) \in c$. Vom nota prin:

- $CTC_{\mathcal{G}}$ mulțimea componentelor tari conexe ale grafului $\mathcal{G}$;
- $CTC_{\mathcal{G}}^{T}$ mulțimea componentelor tari conexe terminale ale grafului $\mathcal{G}$;
- $v \in V$, prin $c_v \in CTC_{\mathcal{G}}$ ne referim la componenta tare conexă în care apare nodul v, din graful $\mathcal{G}$.

Mulțimea componentelor tari conexe $CTC_{\mathcal{G}}$ formează o partiție a nudurilor grafului $\mathcal{G}$.

Definiție 4. Un graf orientat $\mathcal{G}^* = (V^*, A^*)$ este $graf - CTC$ al grafului $\mathcal{G} = (V, A)$ dacă și numai dacă:

- $V^* = CTC_{\mathcal{G}}$
- $A^* = \{a \in A | c_{s(a)} \neq c_{d(a)}\}$

Propoziție 1. Fie $\mathcal{G} = (V, A)$ un graf orientat, astfel încat:

- $graful - CTC$ corespunzător lui $\mathcal{G}$ este aciclic
- V finit $\Rightarrow CTC_{\mathcal{G}}$ este o mulțime finită ce componente tari conexe
- V infinit $\Rightarrow \forall c_1 \in CTC_{\mathcal{G}}, \exists c_2 \in CTC_{\mathcal{G}}^{T} : DF(c_1, c_2) \neq \emptyset$
- $\forall v_1, v_2 \in V : DF(v_1, v_2) \neq \emptyset \Leftrightarrow DF(c_{v_1}, c_{v_2}) \neq \emptyset$

Propoziție 2. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, $\mathcal{GA}$ graful său de accesibilitate, $M_1, M_2 \in [M_0\rangle$, atunci au loc:

- $M_2 \in [M_1\rangle \Leftrightarrow DF(c_{M_1}, c_{M_2}) \neq \emptyset$
- $M_2 \in [M_1\rangle \Leftarrow |CTC_{\mathcal{GA}}| = 1$

Prin $\mathcal{MA}$ vom nota mulțimea marcărilor acasă ale lui γ

Propoziție 3. Fie $\gamma = (N, M_0)$ o rețea Petri marcată, $\mathcal{GA}$ graful său de accesibilitate, $X \subseteq [M_0\rangle$ și $M \in [M_0\rangle$, atunci au loc:

- $M \in \mathcal{MA} \Leftrightarrow CTC_{\mathcal{GA}}^{T} = \{c_M\}$
- $\mathcal{MA} \neq \emptyset \Leftrightarrow |CTC_{\mathcal{GA}}^{T}| = 1$
- $M_0 \in \mathcal{MA} \Leftrightarrow |CTC_{\mathcal{GA}}| = 1$

Pentru a putea analiza corectitudinea unei rețele Workflow, vom studia mai întâi dacă rețeaua este mărginită, o este marcare acasă și pseudo-viabilitatea rețelei, astfel:

- se construiește arborele de acoperire al rețelei Workflow;

- dacă rețeaua este nemărginită, din Lema 3 se deduce că fluxul de lucru nu este corect;
- dacă rețeaua este mărginită, atunci:
  - verificăm dacă o este marcare acasă fie folosind definiția, fie construind componentele tari conexe terminale ale grafului de accesibilitate și verificăm existența unei singure astfel de componentă și o să îi aparțină (conform Propoziției 3);
  - verificăm pseudo-viabilitatea încercând să găsim toate tranzițiile $t \in T$ în graful de accesibilitate;
  - dacă rețeaua este pseudo-viabilă și o este marcare acasă, atunci putem spune că rețeaua Workflow este corectă.

2 Aplicația "Workflow Management"

Aplicația "Workflow Management" oferă posibilitatea de modelare, analizare și execuție a fluxurilor de lucru, folosind rețele Petri. Față de alte sisteme de administrare a fluxurilor de lucru, aceasta are o interfață disponibilă într-un browser web, iar modelarea și analizarea fluxurilor de lucru nu necesită cunoștințe despre rețelele Petri. În timpul execuției unui flux de lucru, se asigură execuția acțiunilor în ordinea specificată și distribușia automată a resurselor necesare îndeplinirii acțiunilor. Pentru o modularitate crescută și o legătură slabă între cea ce vede utilizatorul și logica de analiză a fluxurilor de lucru, automatizarea execuției, asignarea de resurse, etc. aplicația este împărțită în aplicație web client și aplicație server, comunicarea dintre cele două fiind făcută prin intermediul unui API REST.

În continuarea acestui capitol vom prezenta mai detaliat funcționalitățile aplicației și principalele module(în subcapitolul 1). Subcapitolul 2 va fi axat pe arhitectura aplicației, prezentând tehnologiile utilizate, structurarea mai detaliată a aplicației și unele detalii de implementare.

2.1 Funcționalități

Interfața aplicației fiind în sine o aplicație web la nivel de client, aceasta are o portabilitate foarte mare, nu necesită instalarea locală, astfel făcând mult mai anevoioasă schimbarea mașinii pe care lucrează utilizatorul, el având nevoie doar de un browser web, cum ar fi Chrome, Firefox, Vivaldi, etc.

Angajații (resursele) sunt împărțiți pe roluri, cum ar fi: manager, developer, secretar, administrator, etc. Toți angajații au acces la pagina "Tasks", unde vor vedea acțiunile pe care trebuie să le execute, în funcție de fluxul de lucru de care aparțin, împărțite pe trei categorii:

- **To Do**: reprezintă acțiunile asignate lor, dar care nu au fost încă puse în execuție;
- **Doing**: reprezintă acțiunile care sunt în execuție;
- **Done**: reprezintă acțiunile a căror execuție s-a încheiat.

Doar angajații cu rolul de administrator vor avea acces pe pagina "Workflows", unde vor putea creea, edita și administra fluxuri de lucru.

Pe pagina "Tasks" angajatul își alege un flux de lucru, pentru care să vizualizeze/progreseze acțiunile distribuite lui, din lista din stânga, pe care scrie numele fiecărui flux de lucru în parte. După ce a selectat fluxul de lucru dorit, vor fi afișate toate acțiunile sale, din fluxul

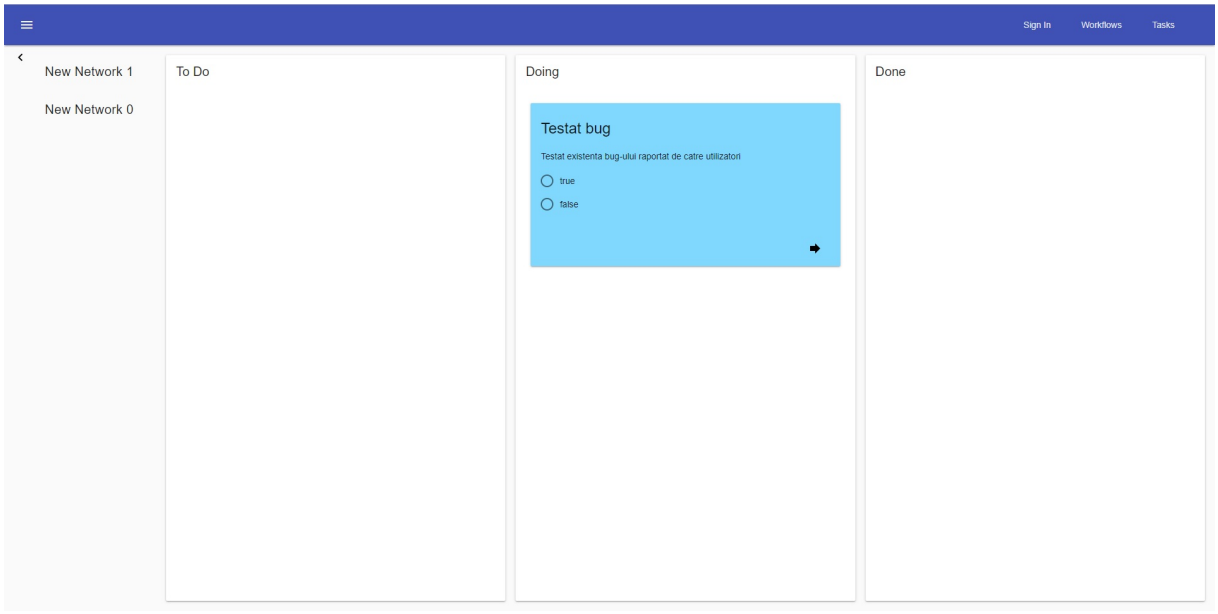

Figura 3: Pagina "Tasks"

de lucru selectat, împărțite pe cele trei categorii. O astfel de acțiune va avea un titlu, dat de numele acțiunii din fluxul de lucru, o descriere și un buton, prin care aceasta va trece din To Do în Doing sau din Doing în Done. Acțiunile din categoria Doing, vor avea în plus posibilitatea de a descărca datele folosite pentru executarea acțiunii, specificate de către administrator, documentele ce trebuie încărcate, pentru a termina execuția acțiunii, la fel, specificate de către administrator, și eventul de ales un răspuns final al acțiunii, dacă specifică administratorul fluxului de lucru.

În pagina "Workflows", administratorul își alege un flux de lucru la care să lucreze, din lista din partea stângă a ecranului, unde sunt trecute numele tuturor fluxurilor de lucru ce aparțin acestuia, sau are posibilitatea de a creea un nou flux de lucru. Fiecare flux de lucru va avea câte un nod de start, denumit "in" și câte un nod de sfârșit, denumit "out". Componenta paginii ce ocupă cel mai mult spațiu și are marginile albastre este zona de modelare a fluxului de lucru. În această zonă utilizatorul va adăuga acțiuni, structuri de control sau dependențe între acestea. Click-ul pe zona de modelare, va îndeplini diverse funcții, depinzând de selecția din meniul din dreapta listei cu fluxurile de lucru:

- **Action**: dacă zona este goală, fără vreun alt nod, va crea o nouă acțiune;
- **Edge**: dacă zona nu conține un nod, atunci se va reseta selecția, altfel:
  - dacă este primul nod pe care s-a dat click, de la ultima resetare a selecției, atunci se va memora ID-ul nodului;
  - dacă este al doilea nod pe care s-a dat click, de la ultima resetare a selecției, atunci se va adăuga un nou arc de la nodul memorat la acesta și se va reseta

selecția;

- **Remove**: dacă zona conține un nod, atunci se va sterge acel nod și se vor șterge și toate arcele interioare (intră în nod) și cele exterioare (ies din nod) acelui nod, dacă zona conține un arc, atunci acesta va fi șters;
- **AND-split**: dacă zona nu conține nici-un alt nod, atunci va fi creat un nou nod de tip AND-split;
- **AND-join**: dacă zona nu conține nici-un alt nod, atunci va fi creat un nou nod de tip AND-join;
- **OR-split**: dacă zona nu conține nici-un alt nod, atunci va fi creat un nou nod de tip OR-split;
- **OR-join**: dacă zona nu conține nici-un alt nod, atunci va fi creat un nou nod de tip OR-join;

Dacă nu este selectată opțiunea **Edge** sau **Remove** și utilizatorul dă click pe o acțiune deja existentă, atunci aceasta va fi selectată și se vor afișa detalii, în partea dreaptă a zonei de modelare, în funcție de starea fluxul de lucru:

- Dacă fluxul este în execuție se vor afișa numele angajaților desemnați acelei acțiuni, împreună cu categoria din care fac parte acțiunile, pentru fiecare dintre ei (To Do, Doing, Done);
- Altfel se vor putea edita numele acțiunii, rolul angajaților ce vor executa acțiunea, ce angajați s-ar prefera pentru executarea acțiunii, datele de intrare și cele de ieșire și descrierea acțiunii;

În momentul în care un flux de lucru este în execuție, în zona de modelare, acesta va fi afișat, doar că acțiunile vor fi colorate, în funcție de categoria în care se află acțiunile în lista fiecărui angajat asignat sau dacă acțiunea poate sau nu intra în execuție. Următoarele culori semnifică:

- **Gri**: Acțiunea nu poate fi încă pusă în execuție;
- **Galben**: Acțiunea poate fi pusă în execuție;
- **Portocaliu**: Acțiunea este în lista "TO Do" pentru toți angajații selectați;
- **Mov**: Acțiunea este în lista "Doing" pentru cel puțin unul din angajații selectați;
- **Verde**: Acțiunea este în lista "Done" pentru toți angajații selectați, semnificând că și-a terminat execuția.

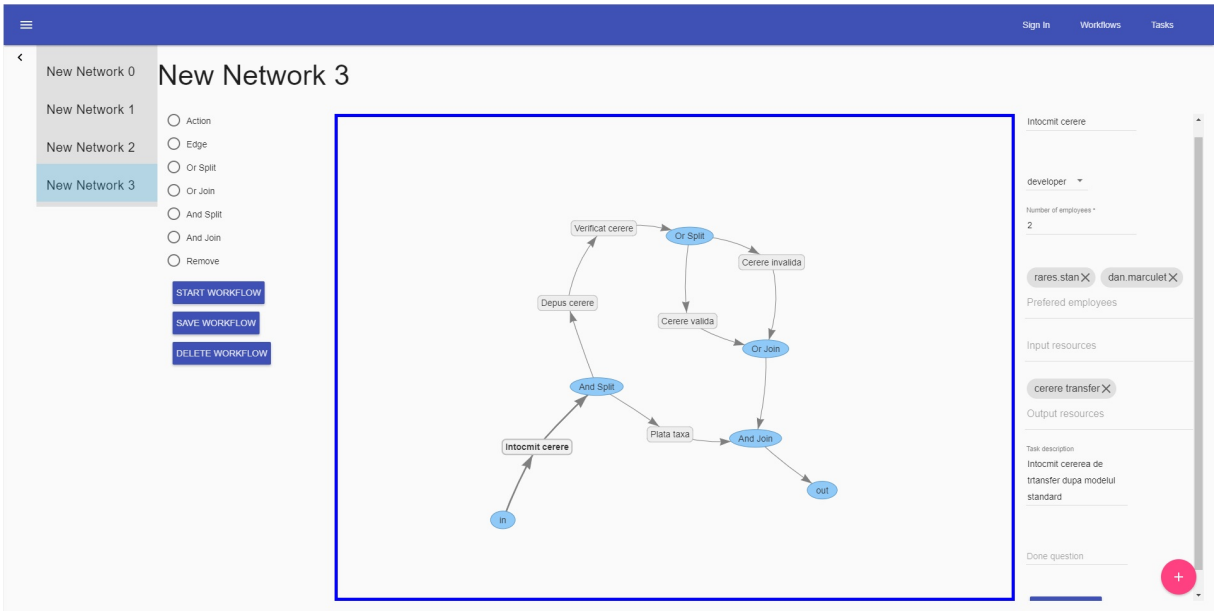

Figura 4: Pagina "Workflows"

Sub meniul ce controlează funcțiile unui click pe zona de modelare, sunt mai multe butoane:

- **Save Workflow**: va salva fluxul de lucru și va verifica corectitudinea acestuia. Dacă fluxul nu este corect se va afișa un mesaj de eroare corespunzător.
- **Start Workflow**: mai întâi va apela salvare fluxului de lucru (făcând și verificările de corectitudine) și va începe execuția fluxului de lucru.
- **Delete Workflow**: va șterge definitiv fluxul de lucru, împreună cu toate acțiunile legate de acesta.

În colțul din dreapta jos este un buton rotund cu un +, acesta va creea un nou flux de lucru ce conține nodurile "in" și "out".

Structurile de control pentru modelarea unui flux de lucru sunt definite și reprezentate grafic ca în Figura 1.

Când un flux de lucru este pus în execuție, acesta este transformat în rețeaua Petri (Workflow) echivalentă, folosind metoda de transformare descrisă în Figura 2. După transformare, rețeaua este verificată, dacă îndeplinește proprietățile necesare ca fluxul de lucru să fie corect. Dacă nu indeplinește acele proprietăți, se va afișa un mesaj de eroare corespunzător. Dacă proprietățile sunt îndeplinite, atunci se vor alege angajați cu rolul specificat de acțiune și li se vor trimite acțiunile ce trebuie să intre în execuție. După ce toți angajații asignați unei acțiuni au terminat execuția acesteia, se va semnala administratorului că se poate avansa fluxul de lucru la urmatoarea stare, pentru care utilizatorul trebuie să specifice următoarea acțiune dorită, dintr-o listă de acțiuni posibile

in acel moment. După ce a selectat acțiunea, se vor asigna acesteai numărul de angajați specificați.

Asignarea angajaților se face automat. Acest proces selectează angajații ce au rolul specificat de acțiune și ia în calcul mai mulți factori:

- **Încărcarea angajaților**: Dacă un angajat are mai puține acțiuni pe care trebuie să le execute, el va avea prioritate mai mare la asignare, față de cineva cu un număr mai mare de acțiuni;
- **Preferințele administratorului**: În timpul modelării fluxului de lucru, administratorul poate specifica o listă de angajați, pe care i-ar prefera pentru execuția unei acțiuni. În momentul asignării angajaților, cei din lista specificată au prioritate mult mai mare față de ceilalți angajați, dar dacă sunt foarte ocupați, aceștia s-ar putea să nu fie selectați, dorindu-se o distribuție cât mai uniformă a angajaților;
- **Experiența angajaților**: Dacă doi angajați au aceați prioritate de asignare pentru o acțiune și a mai rămas un singur loc, se va selecta angajatul cu cea mai multă experiență (vechime în companie).

2.2 Arhitecture aplicației

2.2.1 Tehnologii utilizate

Pentru scrierea aplicației server am ales limbajul Python pentru ușurința adăugării de noi module, ușurința implementării unei aplicații și datorită weak typing-ului. Python fiind un limbaj interpretat, este mai lent decât JAVA sau C/C++, dar cum aplicația trebuie să ruleze în cadrul unei companii, nu necesită procesarea unui număr foarte mare de cereri.

Pentru baza de date am ales Postgresql, datorită suportului implicit al tipului de date JSON, folosit, în principal, pentru stocarea rețelelor Petri și a fluxurilor de lucru.

Modulele de Python folosite sunt:

- **SQLAlchemy**: Oferă posibilitatea de a lega o clasă din Python cu o tabelă din baza de date. Oferă acces foarte ușor la câmpurile legate de o anumită linie din tabel, neavând nevoia de a face explicit join pe tabelele din baza de date și oferă și o abstractizare pentru lucrul cu bazele de date, evitând scrierea de cod specializat pentru modulul de control al bazei de date, făcând astfel baza de date să poată fi schimbată cu mare ușurință;
- **Flask**:Acest modul este un framework ce permite implementarea foarte rapidă și ușoară a unui server web. Pentru implementarea logicii din spatele unui URL, se

specifică URL-ul și parametri (parametri pot să facă parte și din URL, cum ar fi users/<user_id>, user_id ar fi un parametru, iar când am accesa users/3, valoarea parametrului user_id va fi 3) și trebuie creată o funcție fix sub specificarea URL-ului, unde se va afla logica;

- **flask-RESTful**:Este o extensie a framework-ului Flask, ce permite crearea de servicii web de tip REST să fie foarte simpă și ușor de realizat. Pentru a crea logica pentru un URL al serviciului, se derivează din clasa Resource, oferită de flask-RESTful și se implementează acțiunile http la care se dorețte ca serviciul să răspundă (GET, POST, DELETE, etc.). Apoi se adaugă această clasă ca o resursă la un URL, pe care îl specificăm. URL-ul poate conține parametri, ca și în framework-ul Flask. O resursă poate aparține de mai multe URL-uri;

- **flask-sqlalchemy**: Este o extensie a framework-ului Flask, ce permite ca interacțiunea dintre Flask și SQLAlchemy să fie cât mai simplă cu putintă.

Ca și limbaj de programare pentru aplicațiile web la nivel de client, sunt:

- **JavaScript (EcmaScript)**: Un limbaj de scripting de tip weak typing creat special pentru a fi rulat în browser, oferind facilități de modificare a structurii paginii web;

- **Elm**: este un limbaj de programare pur funcțional, ce este transcris în JS (JavaScript), acesta oferă inferență de tipuri, currying-ul funcțiilor, un debuger bazat pe timp și nulte altele;

- **Typescript**: este un limbaj derivat din JavaScript, ce oferă posibilitatea utilizării strong typing-ului.

Am ales dezvoltarea aplicației client în JavaScript, deoarece este limbajul cel mai folosit și este suportat cel mai bine de către AngularJS1.x, iar celelalte limbaje nu ofereau avantaje semnificative.

Ca și tehnologii am folosit:

- **AngularJS1.x**: Este un framework de tip MV* (model, view, whatever), ce mărește gradul de reutilizare a codului, cu ajutorul directivelor, atât la nivel de HTML/CSS, cât și la nivel de JS. Oferă posibilitatea de a simula existența mui multor pagini, fără a apela serverul pentru toate resursele, folosind serviciul de routing. AngularJS oferă o legare automată între șablon (view) și model, ce este referit prin variabila scope, pentru acea pagină. Dacă se schimbă datele din model, șablonul va fi reactualizat automat, pentru a reflecta schimbarea. Directivele pot reprezenta componente unitare sau compuse ale paginii, spre exemplu ele pot fi un singur buton sau chiar

un formular întreg, logica, de verificare a formularului, acșiunile butonului, etc. sunt definite în interiorul directivei, astfel ele având posibilitatea de a fi refolosite și în alte aplicații;

- **AngularMaterial1.x**: Este o suită de directive și servicii adiționale pentru AngularJS1.x, ce oferă posibilitatea de a crea interfețe adaptive pentru diferite tipuri și dimensiuni de ecran. Pe lângă directive și servicii, AngularMaterial1.x oferă aspectul material de la Google, pentru toate directivele sale și un serviciu pentru setarea și editarea temei de culori;

- **VisJS**: Este o bibliotecă ce oferă posibilitatea de a desena și interacționa cu grafuri. Acesta oferă o structură de date proprie, pentru salvarea grafurilor. Acestea sunt reprezentate de două mulțimi: una pentru noduri și alta pentru arce. Interacțiunea cu grafurile constă in capturarea evenimentelor ce la mouse și posibilitatea de a muta nodurile, a da zoom in și zoom out, a muta tot graful stânga/dreapta, sus/jos. Pe lângă desenarea și interacțiunile de mai sus oferă posibilitatea de a aranja automat nodurile astfel încât arcele suprapuse să fie într-un număr cât mai mic.

2.2.2 Structure aplicației

"Workflow Management" este împărțită în două mari aplicații:

- **Aplicația la nivel de client**: Se ocupă cu afișarea fluxului de lucru, interacțiunea cu acțiunile la care a fost selectat utilizatorul și editarea și creerea fluxurilor de lucru;

- **Aplicația server (serviciul web)**:Expune o modalitate de comunicare cu aplicația client, stochează datele, conține logica de reprezentare, verificare, execuție a fluxurilor de lucru cu ajutorul rețelelor Petri și atribuie angajați activităților din fluxurile de lucru.

Aplicația client este împărțită în:

- **Controller-e**: Câte un controller pentru fiecare pagină a aplicației, ce conține logica și datele de afișare ale paginii;

- **directive**: Logica componentelor modulare ce sunt folosite în toată aplicația;

- **servicii**: Colecții de funcții pentru comunicarea http cu serverul dar și cu stocarea locală a datelor (local storage, session storage cât și cookies);

- **Șabloane**: Sunt fișiere html ce sunt folosite pentru afișarea paginilor dar și a directivelor, acestea pot folosi și la rândul lor alte șabloane sau directive.

Controller-ele aplicației sunt pentru două pagini, cea de "Tasks" și cea de "Workflows". TasksController-ul doar are logica pentru selecția unui flux de lucru, iar logica unei unități de lucru este pusă în directiva acesteia. Datele paginii "Tasks" sunt reținute în scopul controller-ului. WorkflowController conține logica pentru evenimentele captate de biblioteca VisJS și logica pentru salvarea, selectarea ștergerea, actualizarea nodurilor și începerea execuției fluxului de lucru.

Componentele modulare ale aplicației sunt:

- **Listele To Do, Doing și Done**: Sunt implementate în directiva KanbanTacks, ce încarcă șablonul KanbanTracks.html, în care sunt definite cele 3 liste și le este specificat și variabila în care se găsesc datele fiecăreia și este creat câte un element TrackItem pentru fiecare element din liste;

- **Elementul unei liste**: Este implementat în directiva TrackItem, în care este încărcat țablonul TrackItem.html și este definită logica de avansare a acțiunilor din To Do în Doing și din Doing în Done. Fiecare element are un titlu, o descriere, eventual o listă de răspunsuri și date de intrare și de ieșire ale fiecărei acțiuni;

- **Detaliile acțiunilor**: Sunt înglobate de directiva NodeDetails, ce încarcă un anumit șablon, dacă fluxul de lucru este în execuție sau nu:

  - **În execuție**: va încărca șablonul RunningNode.html, ce va afișa numele angajatului și lista în care se află această acțiune, pentru acel angajat;
  - **Altfel**: va încărca șablonul EditingNode.html, în care se pot edita toate detaliile acțiunii, cum ar fi: numele acesteia, rolul și numărul angajaților ce vor fi asignați, date de intrare și de ieșire, etc.

Pentru comunicarea cu serverul, sunt implementate două servicii, ce conțin mai multe funcții specializate în comunicarea cu un URL din cele expuse de server. Cele două servicii sunt TaskService și WorkflowService.

TaskService se ocupă cu comunicarea cu serverul doar pentru pagina "Tasks", din aplicația client, și expune următoarele funcții:

- **AdvanceTask**: Accesează URL-ul, folosind acțiunea http POST, /users/ <user_id>/ tasks/ <workflow_id>/<task_id> pentru a avansa acțiunea cu ID-ul task_id, a utilizatorului cu ID-ul user_id, din fluxul de lucru cu ID-ul workflow_id de la To Do în Doing sau de la Doing în Done;

- **GetTasks**: Accesează URL-ul, folosind acțiunea http GET, /users/<user_id>/tasks/ <workflow_id> pentru a accesa listele de acțiuni ale utilizatorului cu Id-ul user_id, din fluxul de lucru cu ID-ul workflow_id;

- **GetParticipatingWorkflows**: Accesează URL-ul, folosind acțiunea http GET, /users/<user_id>/tasks, pentru a accesa lista cu fluxurile de lucru în care, utilizatorul cu ID-ul user_id, a fost selectat pentru cel puțin o acțiune.

WorkflowService se ocupă cu comunicarea cu serverul doar pentru pagina "Workflows", din aplicația client și expune următoarele funcții:

- **CreateNewWorkflow**: Accesează URL-ul, folosind acțiunea http POST, /users/<user_id>/workflows, pentru a crea un nou flux de lucru, pentru administratorul cu ID-ul user_id;
- **GetWorkflowList**: Accesează URL-ul, folosind acțiunea http GET, /users/ <user_id>/workflows, pentru a accesa lista de fluxuri de lucru existente ale administratorului cu ID-ul user_id;
- **GetWorkflow**: Accesează URL-ul, folosind acțiunea http GET, /users/<user_id>/workflows/<workflow_id>, pentru a accesa datele fluxului de lucru cu ID-ul workflow_id, a administratorului cu ID-ul user_id;
- **UpdateWorkflow**: Accesează URL-ul, folosind acțiunea http POST, /users/<user_id>/workflows/<workflow_id>, pentru a actualiza datele fluxului de lucru, cu cele trimise în format JSON, identificat cu ID-ul workflow_id, a administratorului cu ID-ul user_id;
- **StartWorkflow**: Accesează URL-ul, folosind acțiunea http POST, /users/<user_id>/workflows/<workflow_id>/start, pentru a pune în execuție fluxul de lucru cu ID-lu workflow_id, a administratorului cu ID-ul user_id;
- **DeleteWorkflow**: Accesează URL-ul, folosind acțiunea http DELETE, /users/<user_id>/workflows/<workflow_id>, pentru a șterge fluxul de lucru cu ID-ul workflow_id, a administratorului cu ID-ul user_id;
- **RunNextNode**: Accesează URL-ul, folosind acțiunea http POST, users/<user_id>/next-action/<workflow_id>/<next_node_id> pune în execuție acțiunea cu ID-ul next_node_id, din fluxul de lucru cu ID-ul workflow_id, a administratorului cu ID-ul user_id, dacî este posibil.

Comunicarea dintre client și server este făcută cu ajutorul unu API REST, expus de către server. Acesta oferă câteva URL-uri cu diverse funcționalități, înaintea primului "/" se va adăuga numele domeniului la care se află serverul:

- **"/users/<user_id>"**:Este posibilă doar acțiunea http POST, cu ajutorul căreia utilizatorul cu ID-ul user_id se autentifică;

- **"/users/<user_id>/workflows"**: Acceptă doar acțiunea http GET, pentru a returna lista de fluxuri de lucru pe care administratorul cu ID-ul user_id le-a creat;
- **"/users/<user_id>/workflows/<workflow_id>"**: La acțiunea http:
  - **GET**: Va returna fluxul de lucru cu ID-ul workflow_id, creat de administratorul cu ID-ul user_id. Pe lângă fluxul de lucru, se mai trimit și id-ul, numele, rolurile angajaților din companie și starea procesului Workflow;
  - **POST**: Va suprascrie in baza de date, fluxul de lucru cu ID-ul workflow_id, al administratorului cu ID-ul user_id, cu fluxul de lucru primit ca parametru JSON, pe câmpul "workflow", sub forma a două liste, una cu nodurile, iar cealaltă cu arcele fluxului de lucru. Dacă după verificarea fluxului de lucru reiese că acesta nu este corect, atunci se va trimite un răspuns JSON, a cărui câmp error va conține mesajul de eroare cu tot cu eventuale indicații pentru încercarea de a corecta fluxul de lucru;
  - **DELETE**: Va șterge din baza de date fluxul de lucru cu ID-ul workflow_id, împreună cu toate acțiunile dependente de acesta;
- **"/users/<user_id>/workflows/<workflow_id>/start"**: Acceptă doar acțiunea http POST va pune fluxul de lucru, cu ID-ul workflow_id, al administratorului user_id, în execuție. Se va trimite fluxul de lucru actualizat, după punerea acestuia în execuție, modificându-se culorile acțiunilor, pentru a reprezenta diferitele stări ale acestora;
- **"/users/<user_id>/tasks"**: La apelarea cu acțiunea http GET, va returna o listă cu fluxurile de lucru în care participă utilizatorul cu ID-ul user_id (este asignat la cel puțin o acțiune din acel flux de lucru);
- **"/users/<user_id>/tasks/<workflow_id>"**: La acțiunea http GET va returna acțiunile, asignate utilizatorului cu ID-ul user_id, ce aparțin fluxului de lucru cu ID-ul workflow_id, filtrate în 3 liste diferite: To Do, Doing și Done. Fiecare acțiune va avea id-ul, titlul, descrierea, datele de intrare și cele de ieșire și starea sa;
- **"/users/<user_id>/tasks/<workflow_id>/<task_id>"**: La acțiunea http POST, va avansa acțiunea cu ID-ul task_id, ce aparține fluxului de lucru cu ID-ul workflow_id și a fost asignată utilizatorului cu ID-ul user_id. La final, va trimite ca și răspuns noile liste cu acțiunile utilizatorului;
- **"/users/<user_id>/next-action/<workflow_id>/<next_node>"**: Acceptă doar acțiunea http POST și va pune în execuție acțiunea cu Id-ul next_node, ce aparține

de fluxul de lucru cu ID-ul workflow_id și a fost creat de administratorul cu ID-ul user_id, dacă acea acțiune este posibilă din starea curentă a fluxului de lucru de care apaține. Ca și răspuns va trimite fluxul de lucru actualizat.

Pe partea de server aplicația este împărțită în două mari module:

- **Rețele Petri**: ce se ocupă de reprezenatarea, verificarea, transformarea și efectuarea operațiilor asupra rețelelor Petri (mai exact rețelelor Workflow) și conține fișierele:
  - **Petri Net**: Este implementată reprezentarea rețelelor Petri cu ajutorul grafului de accesibilitate, algoritmul de verificare a corectitudinii acestora, folosind componentele tari conexe terminale ale grafului de accesibilitate, pentru determinarea marcărilor acasă și posibilitatea stocării unei stări curente și avansarea acesteia;
  - **Petri Marking**: Stochează datele unei marcări a rețelelor Petri, locațiile și valorile (numărul de puncte) asociate fiecăreia și implementează logica de generare a unei noi marcări, dată marcarea curentă, rețeaua Petri și tranziția ce se produce. Pe lângă acestea oferă abilitatea de compararea a două marcări;
  - **Workflow Data**: Se ocupă de stocarea persistentă a datelor unui flux de lucru, în formă de rețea Workflow, memorând locațiile, nodul de start, nodul de sfârșit, tranzițiile și arcele dintre acestea;
  - **Convert Workflow**: Convertește reprezentarea unui flux de lucru în rețeaua Workflow echivalentă, folosind regulile de transformare din Figura 2;
- **Server**: ce se ocupă de comunicarea cu aplicația la nivel de client, logica de asignare a angajaților, logica pentru avansarea fluxurilor de lucru și stocarea persistentă a datelor. Conține următoarele fișiere:
  - **Server**: Definește API-ul REST și resursele aflate la fiecare URL, printr-o clasă cu metode numite dupa acțiunile http posibile;
  - **Utility Functions**: Implementează o suită de funcții, cu ajutorul cărora este definită logica pentru asignarea angajaților la acțiuni, avansarea execuției unei acțiuni, punerea în execuție a unui flux de lucru și controlul execuției proceselor Workflow. Aceste funcții funt folosite de către resursele API-ului REST;
  - **Db Model**: În care sunt definite clasele pentru stocarea persistentă a datelor cu ajutorul ORM-ului :
    - **Employee**: Fiecare angajat are un id, nume de utilizator, nume și prenume, o parolă, un rol și data când a fost angajat. ORM-ul mai oferă

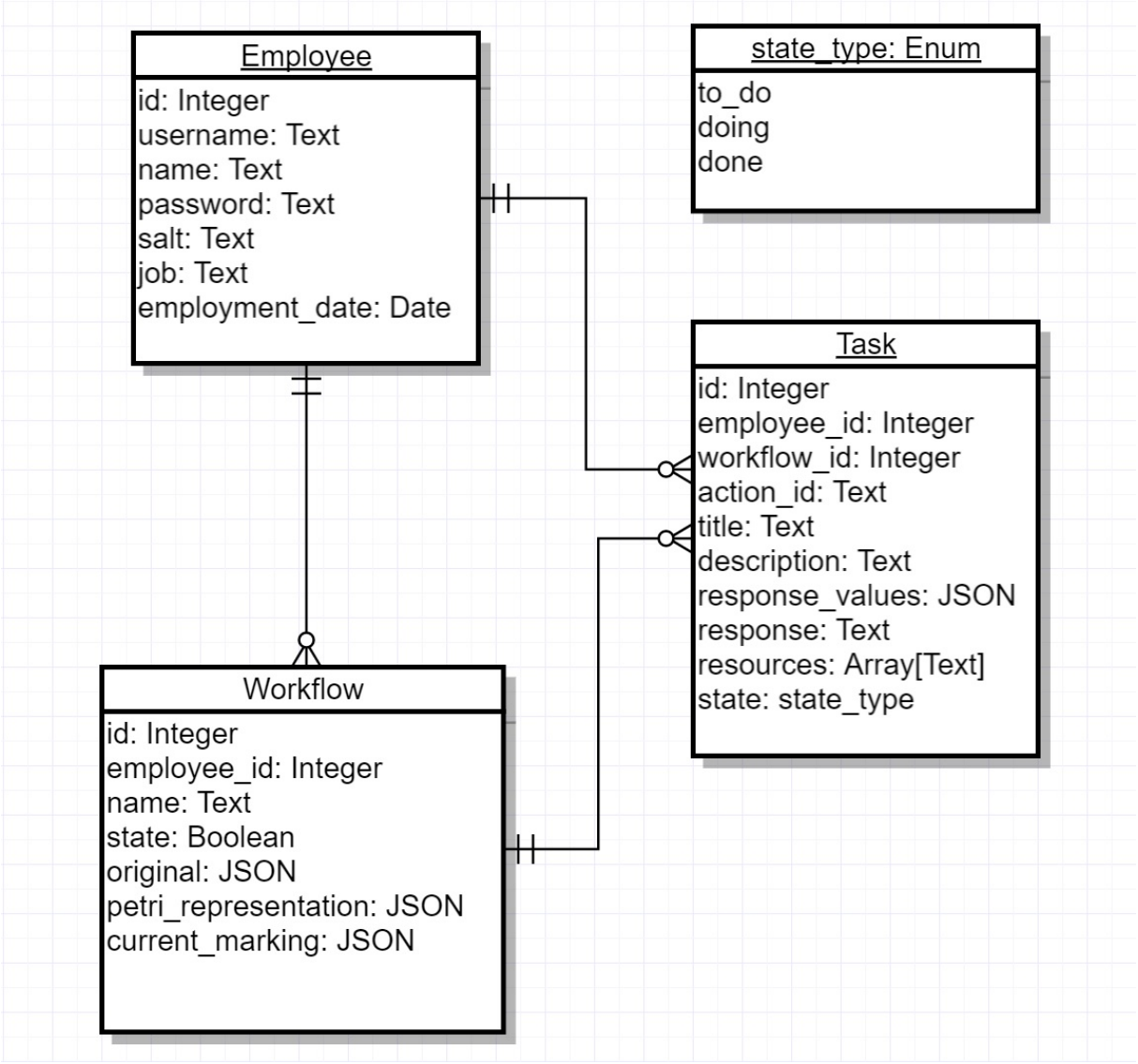

Figura 5: Schema ERM a bazei de date

și acces din instanța unui angajat la fluxurile de lucru și acțiunile le care a fost repartizat, deoarece cheia primară id este cheie străină în tabelele Task și Workflow;

* **Task**: Pentru o acțiune reținem id-ul ei, o cheie străină către angajatul repartizat pentru executarea acțiunii, o cheie străină către fluxul de lucru de care aparține, id-ul acțiunii din reprezentarea grafică a procesului Workflow, titlul, descrierea valorile de răspuns posibile, datele de intrare și de ieșire, starea acțiunii (dacă este în lista de To Do sau în lista Doing sau Done a angajatului) și răspunsul selectat de angajat la terminarea execuției;
* **Workflow**: Pentru fluxurile de lucru reținem id-ul lor, o cheie străină către administratorul ce deține fluxul, numele, starea (dacă este în execuție sau

nu), reprezentarea cu care interacșionează administratorul, reprezentarea sub formă de rețele Workflow, marcarea la care se află execuția rețelei și acțiunile ce depinde de acest flux de lucru, oferite de către ORM.

2.2.3 Detalii de implementare

Rețeaua Petri, marcările sale și reprezentarea unei rețele Workflow ca locații, tranziții și arce au asociate diverse funcții și sunt folosite ca și componente în alte reprezentări, cum ar fi nodurile grafului de accesibilitate a unei rețele Petri, ce sunt marcări acesteia, le-am înglobat în câte o clasă.

Clasa PetriNet construiește graful de accesibilitate, menține starea rețelei și verifică dacă o rețea este corectă, folosind Teorema 1, din Subcapitolul 1.3. Aceasta are ca atribute graful de accesibilitate, graful de accesibilitate transpus, marcarea curentă și reprezentarea rețelei Petri ca locații, tranziții și arce.

Clasa PetriMarking poate generea o nouă marcare, ce se obține dintr-o marcare inițială și o tranziție ce se produce, conține logica de comparare a două marcări. Atributul acesteia este un dicționar, cheile fiind locațiile rețelei Petri, iar valorile numărul de puncte/resurse dintr-o locație.

Clasa WorkflowData stochează o rețea Petri sub formă grafică, ca tranziții, locații și arce, pentru a putea fi convenabil de serializat, pentru baza de date. Această clasă implementează funcții pentru a determina arcele ce conțin o anumită componentă, ca extremitate inițială, finală sau oricare, accesul la o anumită componentă și serializarea rețelei Petri în format JSON. Ca și atribute această clasă are câte o listă de tranziții, locații și arce și locația de start și de sfârșit.

În algoritmul de verificare a corectitudinii folosim graful de accesibilitate atât pentru determinarea carcărilor acasă cât și pentru verificarea pseudo-viabilității. Pentru a contrui graful de accesibilitate, mai întâi trebuie construit graful de acoperire, pentru a evita construcția unui graf de accesibilitate infinit sau în timpul construcției grafului de accesibilitate se verifică dacă există cel putin o marcare a rețelei Petri care acoperă măcar un nod al grafului construit deja. Astfel rezultă algoritmul următor.

Algoritmul 1. Construcția grafului de accesibilitate al unei rețele Workflow

Graful de accesibilitate este reprezentat în memorie ca o listă de adiacență. Datorită faptului că nodurile grafului sunt defapt marcări a rețelei Petri și arcele sunt etichetate, vom folosi un dictionar, a cărui chei sunt marcările rețelei Petri, iar valorile sunt alte dicționare cu cheile fiind etichetele arcelor, iar valorile reprezentând noduri din graf. O marcare a rețelei Petri este implementată folosind un dicționar a carui chei sunt numele unice ale locațiilor, iar valorile acestuia sunt numărul de resurse/puncte în locații. Tabelele hash oferă posibilitatea de a folosi ca și cheie diverse tipuri de date, pe când listele clasice

```
def _create_nodes(self, petri_node: PetriNode):
    if petri_node not in self._graph:
        self._graph[petri_node] = dict()
    if petri_node not in self._graphT:
        self._graphT[petri_node] = dict()

    for transition in self._workflow_data.get_transitions():
        try:
            current_marking = PetriMarking.generate_marking_from(
                petri_node.get_marking(),
                self._workflow_data,
                transition)

            for node in self._graph.keys():
                if current_marking > node.get_marking():
                    print(current_marking)
                    print(node.get_marking())
                    raise InfinitePetriNetException("The Workflow cannot "
                                                    "finish, there will always "
                                                    "be a possible action")

            neighbour = PetriNode(current_marking)
            self._graph[petri_node][transition] = neighbour
            if neighbour not in self._graphT:
                self._graphT[neighbour] = dict()
            self._graphT[neighbour][transition] = petri_node
        except IllegalMarkingException:
            pass

    for transition in self._graph[petri_node].keys():
        if self._graph[petri_node][transition] not in self._graph:
            self._create_nodes(self._graph[petri_node][transition])
```

Figura 6: Implementarea construcției grafului de accesibilitate

folosesc doar valori de tip int și pe lângă aceasta, dicționarele oferă o complexitate timp aproximativ constantă, pentru accesarea valorilor, știind cheia.

Pasul 1: construim marcarea inițială a rețelei Workflow (ce va avea valoarea 1 pentru locația i și 0 în rest);

Pasul 2: contruim recursiv graful de accesibilitate, porning de la o marcare M, astfel:

- pentru fiecare tranziție $t \in T$ încercăm să construim o nouă marcare M^*, pornind de la marcarea M primită ca argument. Dacă M^* este o marcare validă, atunci:
  - verificăm existența unui nod N în graful de accesibilitate construit momentan pentru care $M^* > N$. Dacă există un astfel de nod N, atunci rețeaua este nemărginită $\Rightarrow$ graful de accesibilitate va fi infinit și rețeaua

Workflow nu va fi corectă, conform Teoremei 1 din Subcapitolul 1.3, și vom opri construcția grafului de accesibilitate;

- pentru fiecare marcare M^*, adăugam un arc marcarea M la marcarea M^*, în graful de accesibilitate, având eticheta t;

- pentru fiecare marcare M^* generată precedent, ce nu se găsea anterior în graf, reluăm Pasul 2.

Acest algoritm este implementat în funcția _create_nodes, din fișierul petri_net.py. Codul pentru construcția unei noi marcări este implementat în funcția generate_marking_from din fișierul petri_marking.py și primește ca argumente marcarea curentă, rețeaua Petri și tranziția ce se va executa și returnează o nouă marcare, obținută prin aplicarea tranziției la marcarea primită ca argument. Dacă in procesul generării noii marcări, se obține o marcare ce are pe toate locațiile 0 sau în cel puțin o locație o valoare negativă, se va arunca excepția IllegalMarkingException.

Verificarea corectitudinii unei rețele Workflow constă în determinarea pseudo-viabilității rețelei și apartenența marcării o în mulțimea marcărilor acasă. Algoritmul de verificare a corectitudiniieste format din două părți:

- verificarea pseudo-viabilității rețelei Workflow, verificând apariția tuturor tranzițiilor rețelei Workflow în graful de accesibilitate;

- determinarea mulțimii de marcări acasă și verificarea apartenenței marcarii o acestei mulțimi.

Algoritmul 2. Determinarea mulțimii marcărilor acasă

Pasul 1: Determinarea componentelor tari conexe, folosind algoritmul lui Kosaraju ([10]);

Pasul 2: Determinăm componentele tari conexe terminale astfel: pentru fiecare componentă tare conexă verificăm dacă din toate nodurile acesteia pleacă arce spre noduri din aceași componentă tare conexă;

Pasul 3: Dacă numărul de componente tari conexe terminale este diferit de 1 atunci nu există marcări acasă. Dacă există marcări acasă, acestea sunt nodurile singurei componente tari conexe terminale.

Algoritmul 3. Algoritmul lui Kosaraju, pentru determinarea componentelor tari conexe ale unui graf orientat

Algoritmul folosește două parcurgeri în adâncime. Prima parcurgere în adâncime este folosită pentru a determina ordinea nodurilor de start pentru a doua parcurgere. A doua

```
def _strongly_connected_components(self):
    self._used = dict.fromkeys(self._graph.keys(), 0)
    self._where = dict.fromkeys(self._graph.keys(), -1)
    self._discoverd = []
    SCC = dict()

    for node in self._graph.keys():
        if self._used[node] == 0:
            self._DFP(node)

    count = 0
    for node in reversed(self._discoverd):
        if self._where[node] == -1:
            self._DFM(node, count)
            count += 1

    for node in self._graph.keys():
        if self._where[node] not in SCC.keys():
            SCC[self._where[node]] = []
        SCC[self._where[node]] += [node]

    return SCC

def _DFP(self, node):
    self._used[node] = 1
    for neighbour in self._graph[node].values():
        if self._used[neighbour] == 0:
            self._DFP(neighbour)
    self._discoverd.append(node)

def _DFM(self, node, count):
    self._where[node] = count
    for neighbour in self._graphT[node].values():
        if self._where[neighbour] == -1:
            self._DFM(neighbour, count)
```

Figura 7: Implementarea algoritmului lui Kosaraju

parcurgere în adâncime este folosită pentru determinarea efectivă a componentelor tari conexe.

La începutul algoritmului apelăm, pentru fiecare nod ce nu a fost vizitat, parcurgerea în adâncime, pe graful inițial și la întoarcerea din recursie adaugă nodul curent într-o stivă, ce este simulată cu ajutorul unei liste. După ce au fost vizitate toate nodurile, putem începe determinarea componentelor tari conexe, folosind ordinea acestora din stivă, parcurgând lista în ordine inversă. Astfel, pentru fiecare nod din stivă, ce nu a fost asignat într-o componentă tare conexă, creem o nouă componentă ce îl va conține, deocamdată doar pe el și apoi parcurgem graful transpus, în adâncime, folosind nodul curent ca nod de start și adăugăm toate nodurile vizitate acestei noi componente tari conexe.

Pentru a nu construi în timpul execuției acestui algoritm graful de accesibilitate

transpus, acesta este construit în același timp cu graful normal.

Pentru că nodurile din graf sunt defapt marcări ale rețelei Petri, folosim un dicționar (o tabelă hash) pentru a determina dacă un nod a fost sau nu vizitat, pentru prima parcurgere în adâncime și un al doilea dicționar, pentru a reține componenta tare conexă în care se află un nod, astfel cheile dicționarului sunt ID-urile componentelor tari conexe, iar valorile sunt liste de marcări a rețelei Petri.

Complexitatea timp al acestui algoritm este $\mathcal{O}(N + M)$, unde N este numărul de noduri ale grafului, iar M numărul de arce ale grafului.

Acest algoritm se găsește în fișierul petri_net.py, în funcția _strongly_connected_components.

Rețeaua Workflow echivalentă a unui flux de lucru este obținută cu ajutorul următorului algoritm, folosind schema de transformare din Imaginea 2, Subcapitolul 1.3:

Algoritmul 4. Obținerea rețelei Workflow echivalente

Pasul 1: Acțiunile din fluxul de lucru devin tranziții, iar nodurile *in* și *out* locații;

Pasul 2: Pentru fiecare tranziție adăugăm o locație și modificăm extremitatea inițială a tuturor arcelor ce pleacă din acea tranziție, cu locația nou adăugată;

Pasul 3: Pentru restul nodurilor le transformăm astfel, luând în considerare faptul că fiecate acțiune este urmată de căte o locație, adăugate la **Pasul 2**:

- **AND-split**: trebuie să mai adăugăm doar o tranziție auxiliară și câte o locație înaintea acțiunilor fix următoare și să le legăm astfel:
  - de la locația ce era conectată de nodul AND-split la tranziția auxiliară;
  - de la fiecare locație adăugată la câte una dintre locațiile următoare;
  - de la tranziția auxiliară la toate locațiile noi;
- **AND-join**: trebuie adăugată o tranziție auxiliară și o locație înaintea acțiunii următoare, iar arcele vor fi:
  - de la cate o locație din cele ce erau unite cu nodul AND-join la tranziția auxiliară;
  - de la noua locație la acțiunea următoare;
  - de la tranziția auxiliară la noua locație
- **OR-split**: pentru fiecare acțiune ce urmează după nodul OR-split adăugăm câte o tranziție, o locație și un arc de la noua tranziție la noua locație și unul de la noua locație la o acțiune. La final de la locația ce era unită cu nodul OR-split, ducem arce spre toate tranzițiile noi;

- **OR-join**: pentru fiecare locație ce era legată de nodul OR-join adăugăm câte o tranziție și un arc de la locație la tranziție. La final adăugă o nouă locație, inaintea acțiunii ce urma după nodul OR-join și arce de la tranzițiile noi la această locație și de la locație la acțiunea următoare.

Algoritmul are ca date de intrare un JSON (în python reprezentarea JSON-urilor ca obiecte este făcută cu ajutorul dicționarelor) cu reprezentarea unui flux de lucru. Reprezentarea rețelei Petri este stocată tot într-un JSON. Pentru primul pas se scot nodurile *in* și *out* din lista de noduri și sunt puse în lista de locații, lista de noduri devenind listă de tranziții.

Algoritmul este implementat în fișierul convert_workflow.py, în funcția convert_to _petri_representation.

Algoritmul 5. Algoritmul de alocare a angajaților Se selectează toți angajații din baza

```
def find_users_for_task(user_type, number, preferred_users):
    all_user_type = [[user, 1000] for user in db_model.Employee.query.filter_by(job=user_type).all()]
    today = datetime.datetime.utcnow()
    for user in all_user_type:
        total_tasks = len(db_model.Task.query.filter(
            db_model.Task.employee_id == user[0].id,
            db_model.Task.state != db_model.StateTypes.done
        ).all())
        user[1] -= 100 * total_tasks
        user[1] += 10 * (
            (today.year - user[0].employment_data.year) * 12 +
            today.month - user[0].employment_data.month
        )
        if user[0].username in preferred_users:
            user[1] += 300
    all_user_type.sort(key=lambda u: u[1], reverse=True)
    all_user_type = list(zip(*all_user_type))[0]
    if number < len(all_user_type):
        return all_user_type[:number]
    return all_user_type
```

Figura 8: Implementarea algoritmului de asignare a angajaților

de date ce au rolul specificat în acțiune și calculează un scor pentru fiecare, astfel:

- Se începe cu scorul de 1000 de pucte;
- Se scad câte 100 de puncte pentru fiecare acțiune neterminată, pentru care a fost selectat deja;
- Pentru fiecare lună de vechime în firmă se adaugă câte 10 puncte;
- Dacă administratorul preferă ca angajatul să fie alocat acestei acțiuni, i se adaugă 300 de puncte.

La final se sortează toți angajații după punctajul obținut și sunt alocați cei cu punctajul maxim.

Parolele utilizatorilor sunt stochate în baza de date sub formă de hash + salt. Acesta este obținut concatenând un șir de 512 biți random la finalul parolei, acesta formând salt-ul, iar parola+salt sunt trimise ca argument funcției hash SHA512. Pentru verificarea unei parole se apelează funcția hash SHA512 pentru parola întrodusă la care concatenăm salt-ul și verifică dacă rezultatul obținut coincide cu parola stocată în baza de date.

Concluzii

Obiectivul acestei lucrări a fost implementarea unui sistem de administrare a fluxurilor de lucru, ce permite modelarea, analizarea și execuția proceselor workflow, în cadrul unei companii, într-o manieră cât mai accesibilă și intuitivă.

Aplicația "Workflow Management" oferă o interfață web ușor de folosit, cu o vizualizare grafică a fluxului de lucru. Datele și resursele necerare execuției sunt specificate la nivelul acțiunii. Sistemul verifică corectitudinea fluxului de lucru, pentru a asigura: terminarea cu succes, lipsa blocajelor și posibilitatea executării tuturor acțiunilo, la un moment dat. Verificarea este făcută cu ajutorul rețelei Workflow echivalente. Eventualele erori, ce duc la un proces Workflow incorect, sunt menționate administratorului, pentru a putea fi remediate. Momentul punerii în execuție a unei acțiuni este decis de către utilizator, sistemul asigurând începerea execuției doar a acțiunilor posibile la un moment dat. Angajații sunt selectați pentru execuția unei acțiuni pe baza mai multor criterii.

Au fost prezentate principalele noțiuni teoretice referitoare la rețele Petri și Workflow, apoi au urmat funcționalitățile aplicației și structura acesteia.

Ca și direcții viitoare aș dori extinderea sistemului astfel încât să se poată folosi și alte tipuri de resurse și posibilitatea specificării multiplelor tipuri de resurse, înafară de resurse umane și îmbunătățirea detecției sursei erorilor.

Bibliografie

[1] S. A. White, "Business process modeling notation (bpmn) version 1.0," *Business Process Management Initiative, BPMI.org*, May 2004.

[2] N. Palmer, *Encyclopedia of Database Systems*, ch. XML Process Definition Language, p. 3601. Springer US, 2009.

[3] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. N. de la Hidalga, M. P. B. Vargas, S. Sufi, and C. Goble, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic Acids Research*, 2013.

[4] Bonitasoft, "Bonita bpm," *Research Gate*, 2013.

[5] Y. Wang and Y. Fan, eds., *Using Temporal Logics for Modeling and Analysis of Workflows*, In proceedings of IEEE International Conference on ECommerce Technology for Dynamic E-Business (CEC-East'04), 2004.

[6] F. Puhlmann and M. Weske, "Using the pi-calculus for formalizing workflow patterns," *Lecture Notes in Computer Science*, vol. 3649, pp. 153–168, 2005.

[7] S. Stupnikov, L. Kalinichenko, and J. Dong, eds., *Applying CSP-like Workflow Process Specifications for their Refinement in AMN by Pre-existing Workflows.*, Proceedings of (ADBIS2002), Sept. 2002.

[8] W. M. P. van der Aalst, *Verification of Workflow Nets*. 1997.

[9] K. Salimifard and M. Wright, *Petri net-based modeling of workflow systems: An overview*. European Journal of Operational Research, 1999.

[10] T. H. Cormen, C. E. Leiserson, and L. Ronald, *Introducere în Algoritmi*, ch. 23 Algoritmi elementari pe grafuri.