OETPN

(Object Enhanced Time Petri Nets)

Rețele Petri înzestrate de timp cu obiecte

1. **Justificare**
2. **Definiții**
3. **Proprietăți**
4. **Exemple de modele OETPN**
5. **Utilitatea și utilizarea modelelor OETPN**
6. **Calitatea modelelor OETPN și calitatea programelor**

1. Justificare

Modelele OETPN conservă posibilitățile de descriere și verificare ale rețelelor Petri clasice, dar au și trăsături noi. Modelele OETPN pot descrie:

- Operații logice (ȘI, SAU, SI-NU, SAU-NU, SAU exclusiv etc.)
- Excludere reciprocă (mutuală), sincronizări
- Utilizarea în comun a resurselor limitate
- Deadlock (blocaj)
- Execuții fără sfârșit
- Arce inhibitoare
- Arce de resetare
- Structura datelor
- Execuții condiționate
- Structuri și execuții distribuite
- Structuri și execuții ierarhizate
- Concurență
- Crearea si distrugerea obiectelor pasive
- Crearea și distrugerea firelor (adică obiectelor active)
- Migrarea taskurilor

2. Definiții

Fig. 1. OETPN

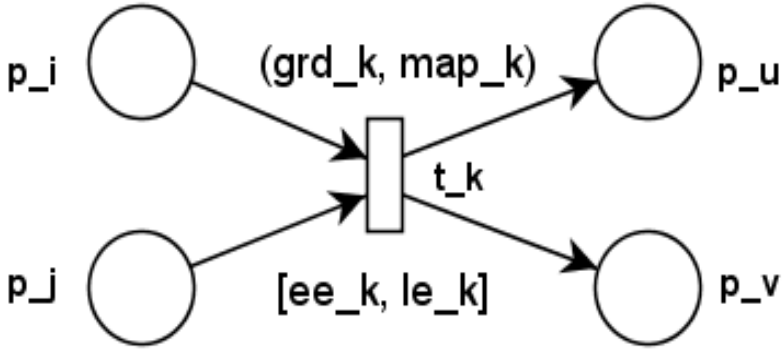


Figura 1 reprezintă o rețea OETPN având:

- tranziție t_k
- două locați de intrare
- două locații de ieșire.
- *grd* condiția de gardă și
- *map* mappingul tranziției.

Un model OETPN este un n-uplu:

$$OETPN = (P, T, pre, post, Inp, Out, D, \delta, Types, type, Grd, Map, \Lambda, M, init, end)$$

unde:

- $P=\{p_1, p_2, \ldots, p_m\}$, $(m \geq 0)$ *este setul finit al locațiilor*,
- $T = \{t_1, t_2, \ldots, t_n\}$, $(n \geq 0)$ *este setul finit al tranzițiilor*,
- *pre*: P × T → {0,1} (cu 0 și 1 numere naturale), este funcția de incidență "înainte" sau de intrare (engl: forward incidence fuction),
- *post* T × P → {0,1} este funcția de incidență "după" sau de ieșire (engl.: backward incidence function),
- $Inp = \{pi_1, \ldots\} \subset$ P, este un set de canale de intrare care inserează jetoane în rețea,
- $Out = \{po_1, \ldots\} \subset$ P, este un set de canale de ieșire care extrage jetoanele din rețea și le transmite mai departe,

- $D = \{\ldots, d_i, \ldots, d_j, \ldots\} \subset \mathbb{N} \times \mathbb{N}$ este un set reprezentând intervalele de întârziere a tranzițiilor, ele fiind atribuite tranzițiilor de aplicația
- δ: $\{t \in T \| {}^{o}t| = 1\} \rightarrow D$; $d_i = \delta(t_i) = [eet_i, let_i]$ - intervalul de timp al tranziției $t_i \in T$ însemnând momentul de timp cel mai devreme al execuției (engl.: *earliest execution time*) și respectiv momentul de timp cel mai târziu al execuției (engl.: *latest execution time*); când $eet_i = let_i$ tranziția t_i este temporizată exact,
- *Types* = {$Class_1$, $Class_2$, …} reprezintă setul claselor software ale obiectelor (jetoanelor) din rețea,
- *type*: $P \rightarrow Types$ este o aplicație care atribuie fiecărei locații *p* o clasă (adică tip) unde *type(p)* este tipul locației *p*; jetoanele introduse în acea locație sunt de tipul menționat,

- **Grd** and **Map** sunt seturi de aplicații de gardă (adică sunt condiții de execuție ale tranzițiilor pe baza locațiilor de intrare) și respectiv de creare (mapare) a jetoanelor în locațiile de ieșire (engl.: *net guard and mapping*),
  - $\Lambda = \{\lambda_t | t \in T, \lambda_t \subseteq grd_t \times map_t\}$ este setul relațiilor dintre condițiile de gardă și mapping ale tranzițiilor cu $\lambda_t = \{(grd_t^i, map_t^i)\}$ care atribuie fiecărei tranziții t perechi compuse dintr-o gardă și o aplicație din seturile grd_t și map_t. Perechile (grd, map) reprezintă *regulile de evoluție* (engl.: rule of evolution) ale modelului.
- **M** este vectorul marcajelor rețelei cu *M(p)* marcajul locației *p* care identifică jetonul obiect de tipul *type(p)*.
- În implementare *M*(*p*) conține un pointer spre un obiect de tipul *type(p)* sau este ϕ (adică, null, nimic sau vid). $M(p) = \Phi$ este echivalent cu

$$M(p) = 0$$

din rețelele Petri clasice.

- *init* este o funcție folosită pentru inițializarea marcajului **M** și pentru startarea execuției modelului OETPN; init ia valori din domeniul creatorului și injectează jetoane în locațiile (adică, marcajul) copilului OETPN.
- $init: Domain(creator) \rightarrow Domain(M)$
- Metoda *init()* este executată de către creatorul OETPN-ului (adică de către părinte) înainte de startarea execuției copilului OETPN. Se consideră că metoda *main()* este creatoarea primului OETPN al programului.
- *end* este o funcție folosită la terminarea sau oprirea copilului OETPN. Ea extrage informații din marcajul rețelei copil și le introduce în domeniul părintelui. Formal metoda *end()* realizează:

$$end: Domain(M) \rightarrow Domain(creator)$$

- Funcția *end()* poate fi folosită de către părinte ca să oprească execuția copilului sau de către copil ca să încheie execuția sa și să transmită informații părintelui.

Notații:

${}^{o}t = \{p \in P | \, pre(p, t) = 1\}$ (2)

$t^{o} = \{p \in P | \, post(p, t) = 1\}$ (3)

Setul canalelor (locațiilor) de intrare notat cu *Inp* ⊂ P și setul canalelor (locațiilor) de ieșire notat cu *Out* ⊂ P descriu interacțiunea rețelei cu mediul extern ei. O locație poate fi simultan și de intrare și de ieșire.

O condiție de gardă a tranziției t_k notată cu grd_k^i este o aplicație (engl.: mapping):

$$grd_t^k : \prod_{p \in {}^{0}t} (Domain(M(p)) \cup \emptyset) \rightarrow \{true, false\}$$

cu *true* și *false* valori din setul Boolean,

iar ϕ semnificând setul vid. Setul tuturor condițiilor de gardă atribuite unei tranziții t_k este notat cu grd_k, iar **Grd** reprezintă setul tuturor condițiilor de gardă ale rețelei.

Mappingul unei tranziții t_k notat cu map_k^i este:

$$map_t^k : \prod_{p \in {}^{0}t} (Domain(M(p)) \cup \emptyset) \rightarrow \prod_{p \in t^{0}} (Domain\big(M(p)\big) \cup \emptyset)$$

(5)

Admisibilitate și executabilitate

O tranziție este *admisibilă (engl.: enabled)* la momentul curent τ_c dacă există cel puțin un set de valori din domeniile locațiilor ei de intrare care satisface o condiție de gardă asiganată ei.

O tranziție este *executabilă (engl.: fireable)* la un moment dat τ dacă a fost admisibilă la momentul τ_c, iar timpul curent este $\tau = \tau_c + \theta$, cu θ o valoare în intervalul de timp asociat tranziției.

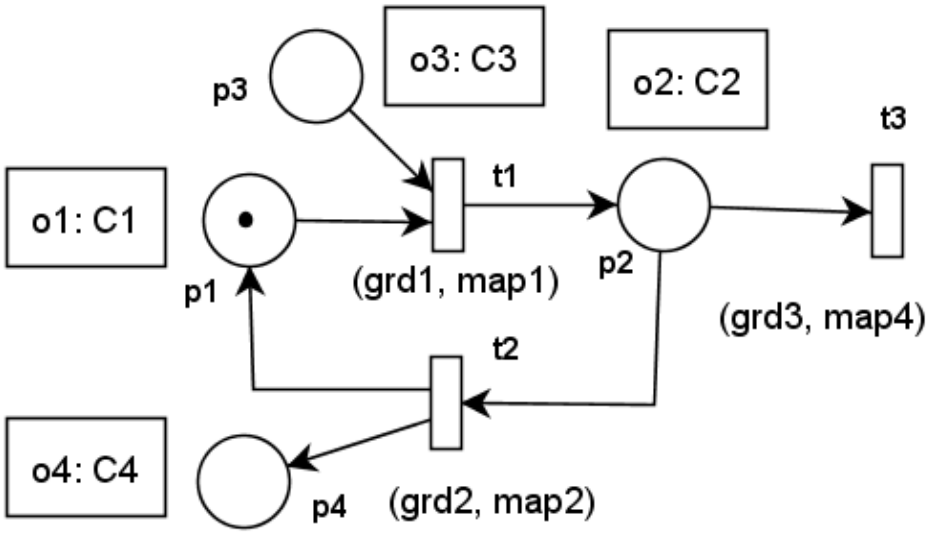
Se presupune că numai o singură condiție de admisibilitate a unei tranziții este îndeplinită la un moment dat, iar aceasta startează execuția tranziției folosind mappingul asociat. Dacă la un moment dat o tranziție poate deveni admisibilă din mai multe condiții de gardă, aceasta poate constitui o nedeterminare, iar prima condiți găsită va starta tranziția.

Descrierea unui model OETPN

OETPN cu jetoanele atașate locațiilor și relațiile (sau regulile) de evoluție asociate tranzițiilor.

- o1, o2, o3 , o4 instanțele asociate locațiilor p1, ..., p4
- C1, C2, C3, C4 sunt clasele (tipurile) asignate locațiilor
- (grd1, map1), (grd2, map2), (grd3, map3) sunt relațiile de evoluție asociate tranzițiilor t1, t2 și t3

Fig. 2. OETPN cu jetoane reprezentate



Tipurile asociate locațiilor sunt din setul acceptat pentru fazele în care sunt utilizate.

Relațiile de evoluție pot folosi metodele *getter* și *setter* definite în clase. Metodele de clasă (declarate *static*) pot de asemenea să fie utilizate.

Conceperea modelelor OETPN

OETPN are diferite niveluri de detaliere în funcție de faza de utilizare în dezvoltarea aplicației.

Pas 1. Stabilirea setului de locații și tranziții împreună cu canalele de intrare și de ieșire

Pas 2. Descrierea rețelei OETPN grafic (echivalent prin matricele *Pre*, Post)

Pas 3. Stabilirea tipurilor locațiilor

Pas 4. Definirea regulilor de evoluție adică a perechilor *(grd, map)*

Pas 5. Stabilirea temporizărilor sau intervalelor de timp asociate tranzițiilor

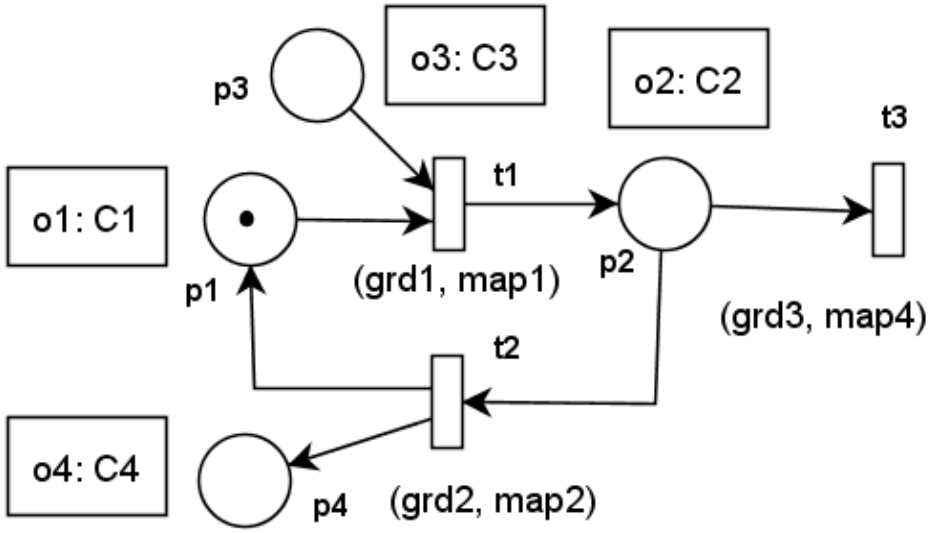
Pas 6. Stabilirea condițiilor inițiale (sau marcajului inițial) M^0.

Deseori este nevoie de reveniri la pași anteriori pentrru completări, eliminări sau modificări.

În faza de specificare nu este nevoie de precizarea exactă a tipurilor jetoanelor și a metodelor (funcțiilor) asociate acestora.

Dacă este nevoie de implementarea modelului de specificare, sunt necesare detaliile.

În faza de implementare este obligatorie declararea exactă a tipurilor locațiilor conform cu limbajul (obiectual) de programare folosit și cu pachetele de clase disponibile.



Aici trebuie definite exact expresiile condițiilor de gardă și ale mappingurilor. Ele trebuie să fie compatibile cu jetoanele la care vor fi asociate.

Modelele OETPN cu intervale de timp sunt utilizate mai des pentru specificare și verificare, în timp ce pentru implementare se utilizează pentru temporizări momente de timp.

Moștenirea sau extinderea (engl.: inheritance or extend)

OOP moștenirea sau extinderea ➔ *extends*

Superclasă ➔ *subclasă*

- *suprascrierea metodelor*
- jetoane = instanțe
- guard & mapping ➔ setter & getter + metode proprii

Detaliile claselor pot fi vag definite în unele faze ale dezvoltării, dar trebuie să fie complet definite pentru faza de implementare.

Extinderea și reducerea rețelelor OETPN

Aceastea nu mai sunt concepte din OOP.

O rețea poate fi extinsă prin adăugarea la structura anterioară a unor noduri și ramuri, sau chiar subrețele.

Pre & Post

Extinderea și reducerea pe:

- orizontală
- verticală

Utilitate:

- dezvoltare
- mentenanță

Sintaxa regulilor de evoluție

Regulile de evoluție asociate unei tranziții t_i având locațiile de intrare $°t_i=\{p_1,p_2\}$ și locațiile de ieșire $t_i°=\{p_3,p_4\}$ se descriu conform sintaxei:

type(m(p_1)= {x: float; y:int} sau *type(m(p_1): {x: float; y:int}*

type(m(p_2)= {x: float; y: float; z:boolean; method(x,y):= (x +2·z)}

type(m(p_3)= {x: float;}

type(m(p_4)= {x: float; y:boolean; z:float;}

grd_i^1:= (m(p_1) ≠ φ and m(p_2) ≠ φ) ➔ map_i^1:= (m(p_3).x= (m(p_1) .y+1.2); m(p_4).y= m(p_2) .z)

grd_i^2:= ((m(p_1).x ≥0) and m(p_2) = φ) ➔ map_i^2:= (m(p_3).x= (m(p_1) .x – 2.0); m(p_4).x= m(p_2) .x)

Când nu există confuzii, se pot utiliza forme prescurtate precum:

x_i = m(p_i).x = p_i.x ; dacă *x* este o variabilă a jetonului din m(p_i).

Relația precedentă (m(p_3).x= (m(p_1) .y+1.2) se poate scrie $x_3 = y_1 + 1.2$.

Condițiile de gardă pot descrie relații logice de tip ȘI (AND), SAU (OR), ȘI-NU (NAND) etc.

Relațiile de tip SAU pot fi scrise prin folosirea mai multor expresii de gardă.

Expresiile condițiilor de gardă și ale mappingurilor trebuie să fie compatibile cu tipurile jetoanelor din locațiile implicate.

Tipurile locațiilor pot conține date de tip primitiv sau de tipul unor clase definite în pachetele mediului în care se va face implementarea, sau definite de dezvoltator.

Metodele de clasă pot fi folosite în expresiile de gardă sau în mappinguri sub forma:

$map_3^3 := (m(p_3).x = m(p_1).x + 10 \cdot Math.random());$

Metodele de instanță din jetoane pot fi folosite sub forma:

$m(p_4).x = m(p_1).x + m(p_2).method(x,y);$

3. Proprietăți ale modelelor OETPN

Rețea Petri marcată (marked net)

O rețea Petri clasică cu un marcaj initial M_0 se numește rețea marcată.

Marcajul = starea rețelei.

OETPN ➔ stare = marcaj + starea jetoanelor active ← pasiv sau în execuție

Starea modelului OETPN reprezintă starea programului și este determinată de:

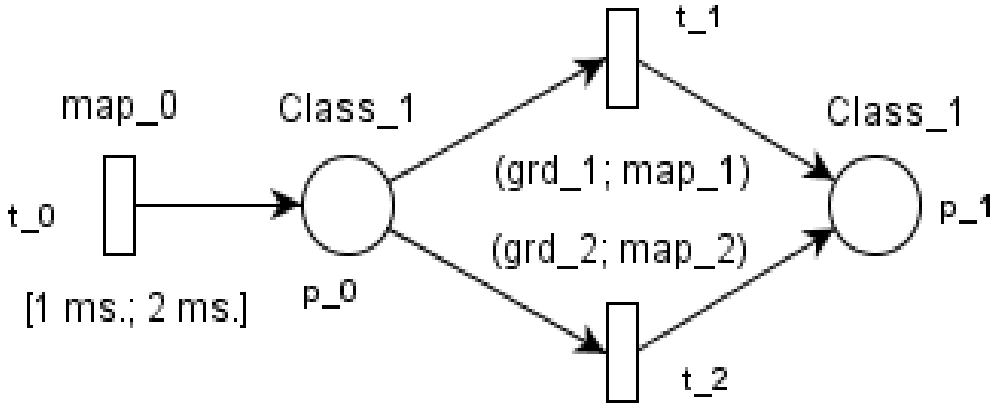
- Obiectele pasive create (sau în curs de creare) și stocate în locații împreună cu valorile pe care le conțin
- Obiectele active create (sau în curs de creare) împreună cu starea lor (activat, oprit, întrerupt, în așteptare, distrrus)
- Starea canalelor de intrare/ieșire (operație realizată, în curs de realizare sau în așteptare)

Selecția

Selecție în OETPN prin:

- tranziții diferite și
- reguli de evoluție multiple
- associate aceleiași tranziții.

Fig. 3. Selecție cu tranziții diferite

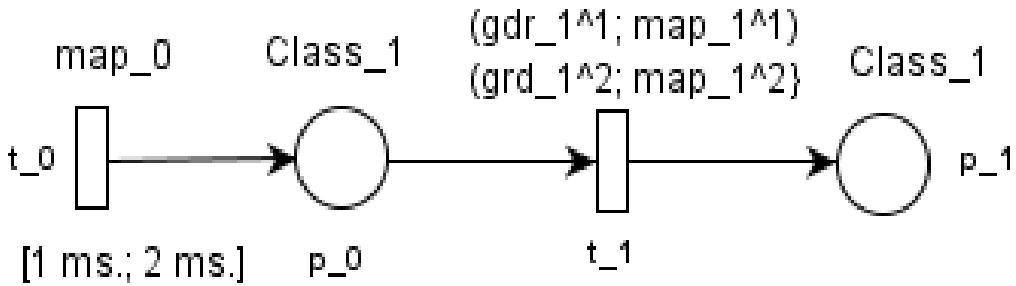


- *p_0, p_1: Class_1 {x:float; y:float}*
- *t_0: {(grd_0:=true; map_0:= (p_1.x=random(); p_1.y=random())}*
- *t_1: {(grd_1:= x>y; map_1:= (p_1.x=p_0.x-p_0.y; p_1.y=p_0.x+p_0.y)}*
- *t_2: {(grd_2:= x<y; map_2:= (p_1.x=p_0.x+p_0.y; p_1.y=p_0.y-p_0.x))}*

Fig. 4. Selecție cu gărzi diferite

Selecția se realizează cu perechi diferite de (grd, map).

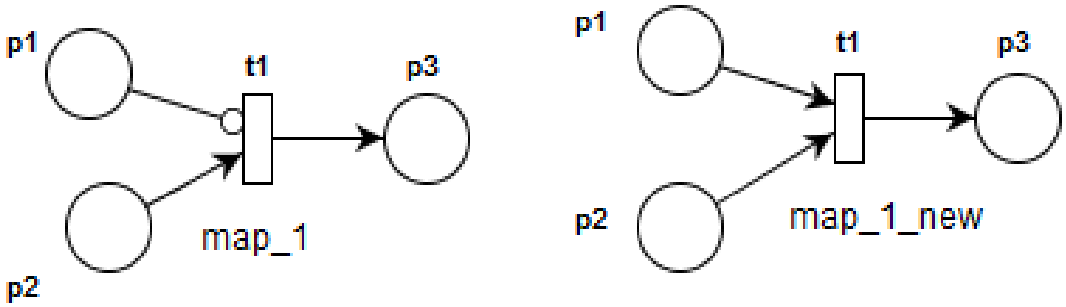
- *grd_1^1 = grd_1; map_1^1 = map_1*
- *grd_1^2 = grd_2; map_1^2 = map_2*



Concluzie: reduce (simplifică) dimensiunea grafului.

Fig. 5. Arce inhibitoare și de resetare

Arce inhibitoare și arce de resetare



a) PN with inhibitor arcs

b) Equivalent UETPN

grd_1:=(p1=φ and p2≠φ).

grd_1:= (p1=φ and p2≠φ) or (p1≠φ and p2≠φ).

Proprietătile introduse aici pot fi folosite la evitarea blocării atunci când aceasta nu se dorește.

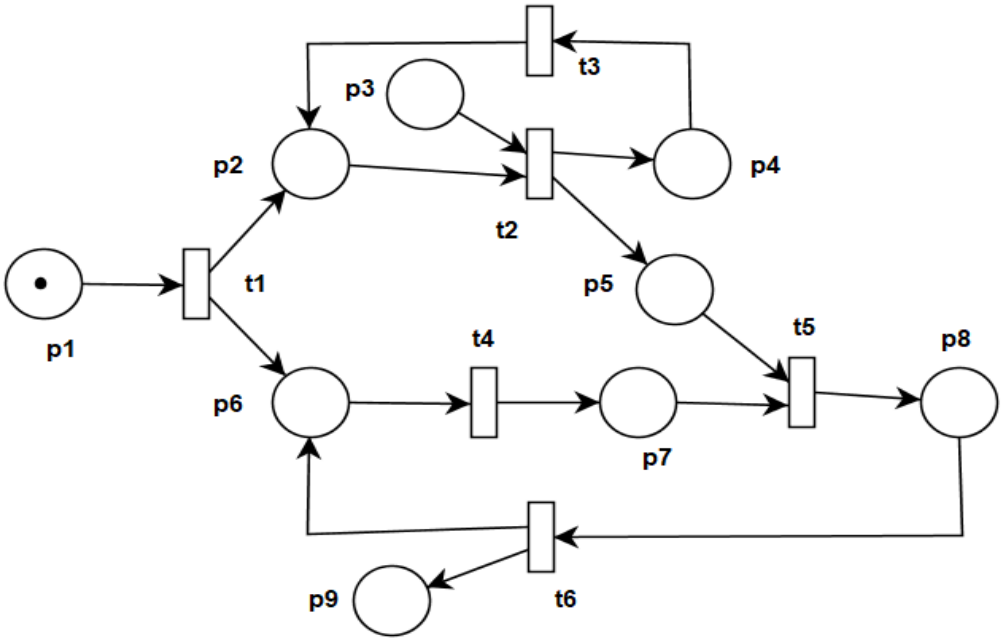
Fig. 6. Taskuri cu sau fără sincronizare

Operații fără blocare, taskuri cu sau fără sincronizare

(engl.: non-blocking operations)

Două taskuri descrise de secvențele:

- t2#t3
- (t4*t5)#t6.



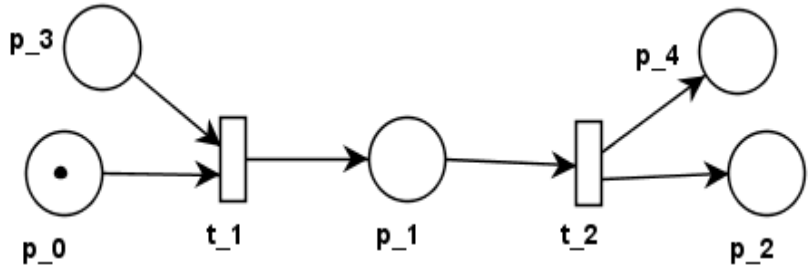
Relațiile dintre cele două taskuri pot fi realizate în două varianate:

- sincronizate când primul task trebuie să execute tranziția t2 înainte ca cel de al doilea să execute tranziția t5
- nesincronizate când cel de al doilea task poate executa t5 și dacă primul nu a executat t2.

Citirea fără blocaj (engl.: non-blocking read)

Fig. 7. Exemplu de citire fără blocare

Inp ={p_3}și Out ={p_4}



Evitarea blocajului prin

t_1: {(grd_1^1:=(p_0≠ φ; and p_3≠ φ), map_1^1), (grd_1^2:=(p_0≠φ and p_3=φ), map_1^2)}

Se specifică:

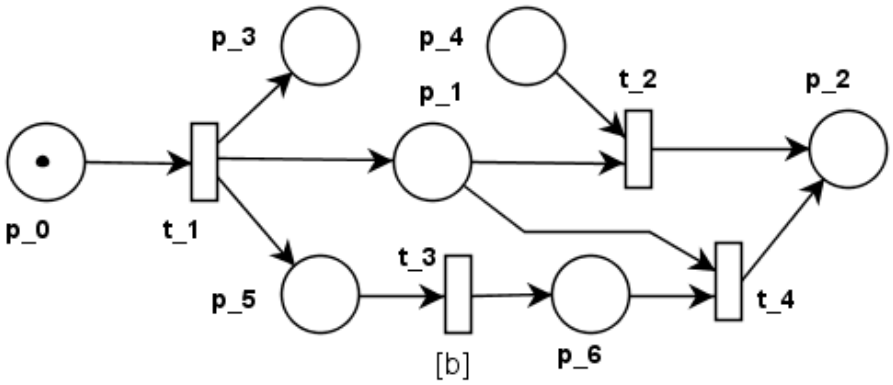
- type(p_3): {x:float}
- type(p_1): {x: float; v: boolean}

Execuția tranziției t_2 ține cont dacă s-a efectuat sau nu citirea executând mappinguri diferite:

t_2: {(grd_2^1:= (p_1.v), map_2^1),
(grd_2^2:= p_1.v=false), map_2^2)}

Fig. 8. Așteptarea limitată a unei operații de citire

Citirea fără blocaj cu așteptare limitată (engl.: bouded blocking read)



Dacă citirea nu se finalizează într-un interval de timp limitat și specifcat (notat cu b), taskul continuă execuția cu altceva.

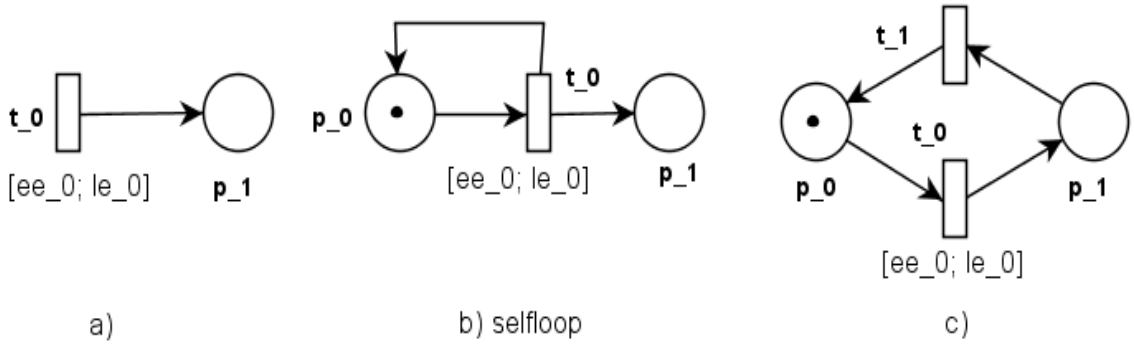
Canalul de ieșire p_3 startează operația de citire prin tranziția t_1 și în același timp setează un temporizator implementat de tranziția t_3. Tranziția t_2 efectuează citirea prin:

(grd_3:= (p_1 ≠ φ and p_4 ≠ φ); map_2)

Dacă nu se poate realiza citirea până se încarcă locația p_6 după *b* unități de timp, se va executa tranziția t_4 marcând în p_2 faptul că nu s-a putut realiza citirea (precum în exemplul precedent).

Bucle și execuții fără sfârșit

Fig. 9. Exemple de execuții infinite



Generează periodic jetoane în locația p_1.

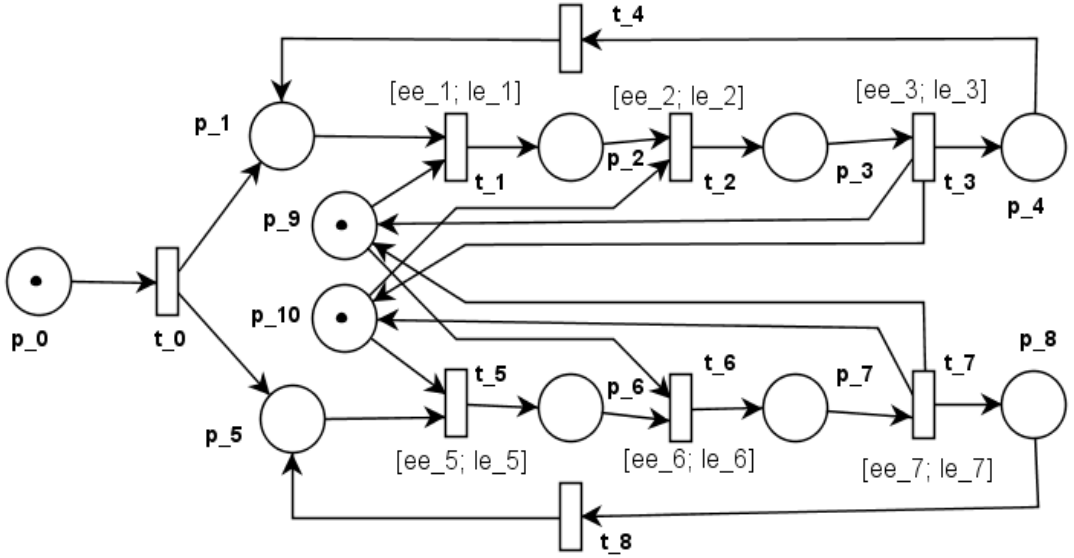
a) poate genera determinist același jeton, sau aleator jetoane diferite.

b) și c) pot genera determinist jetoane diferite luând în considerare marcajul inițial din p_0.

Diferența dintre b) și c) este că c) „consumă” sau poate „consuma” jetonul din p_1.

Deadlock sau blocaj și evitarea blocării

Fig. 10. Deadlock



Modelul corespunde la două taskuri reprezentate de secvențele

σ_1= (t_1*t_2*t_3)#t_4 și

σ_2= (t_5*t_6*t_7)#t_8.

Locațiile p_9 și p_10 reprezintă două resurse achiziționate succesiv (secvențial) de către cele două taskuri. Achiziția succesivă a resurselor poate duce la blocaj (deadlock). Evitarea intrării îun blocaj se poate face prin achiziția simultană a ambelor resurse, dar aceasta duce la reducerea utilizării eficiente a resurselor. Soluția propusă aici implică achiziția celei de adoua resurse numai dacă este liberă (engl.: trylock).

Dacă resursa este ocupată în momentul curent, se renunță la achiziția ei și în plus se eliberează și prima resursă achiziționată.

Implementarea soluției se realizează prin condițiile de gardă ale tranzițiilor t_2*t_3 și respectiv t_6*t_7. Tipul jetoanelor stocate în locațiile p_3 și p_7 conține două variabile booleene v1 și v2 reprezentând achiziția resurselor.

Pentru primul task se asociază tranzițiilor t_2 și t_3:

- t_2:
  - (grd_2^1:= (p_2 ≠ φ and p_10 ≠ φ); map_2^1:= (p_3.v1=true; p_3.v2=true),
  - (grd_2^2:=(p_2 ≠ φ and p_10 = φ); map_2^2:= (p_3.v1=true; p_3.v2=false)
- t_3:
  - (grd_3^1:= (p_3.v1=true and p_3.v2=true); map_3^1:= (utilizează resursele, eliberează resursele)
  - (grd_3^2:= (p_3.v1=true and p_3.v2=false); map_3^1:= (eliberează resursa p_9)

Tranzițiile t_6 și t_7 au asociate perechi (grd, map) similare.

Soluția de mai sus realizează ceea ce se numește în software engineering „roll-back”.

Sifon (engl.: syphon)

Acesta apare în situația când în urma execuției unor tranziții sunt eliminate jetoane, fără a introduce altele necesare, ducând astfel la blocarea pentru totdeauna a unor taskuri (secvențe de tranziții) deoarece lipsesc jetoanele care ar putea să le activeze.

Realizabilitatea (engl.: reachability)

În rețelele Petri clasice se notează cu T^* setul tuturor secvențelor construite cu tranzițiile rețelei , adică $T^* = \{t_0, t_1, ..., t_0{*}t_1, \cdots\}$, cu $\sigma = t_0{*}\ t_1\ {*}\ \cdots$ secvențele de tranziții admisibile, iar cu $M_0[\sigma>M$ marcajul în care ajunge rețeaua pornind din M_0 după execuția secvenței σ. Se notează cu N o rețea Petri și cu

$L(N, M_0) = \{\ \sigma \in T^* |\ M_0[\sigma>\ \}$

limbajul rețelei reprezentâd setul tuturor secvențelor de tranziții executate pornind din marcajul inițial M_0. Se observă că în acest caz sistemul este închis, adică nu este influențat de evenimente exterioare.

Realizabilitatea rețelelor Petri clasice este definită de:

$R(N, M_0) = \{M \in \mathbb{N}^{|P|} |\ \exists\ \sigma \in L(N, M_0), M_0[\sigma>M\ \}$

Realizabilitatea definește setul tuturor marcajelor realizabile (engl.: reachability set) care pot fi atinse (la care se poate ajunge) dintr-o anumită stare.

În OETPN acest concept este mult mai complex datorită faptului că starea rețelei include pe lângă marcaj și valorile variabilelor definite și stocate în jetoanele locațiilor.

Relațiile anterioare devin:

$L(OETPN, S_0) = \{ \sigma \in T^* | S_0[\sigma> \}$

$$R(OETPN, S_0) = \{S \in Domeniul(p_0) \times \cdots \times Domeniul(p_m) | \exists \sigma \in L(OETPN, S_0), S_0[\sigma > S\}$$

Unde S reprezintă starea aplicației care poate fi realizată din starea inițială S_0, iar *Domeniul(p_i)* (i=0,1, ⋯ este domeniul variabilelor din jetoanele stocate în locațiile p_i, i=0, 1, ⋯. Luând în considerare că dintre locațiile rețelei, unele sunt locații de intrare, evoluția modelului este determinată și de valorile aplicate (primite) prin intermendiul canalelor de intrare.

Fie $\mathbf{u}(\tau)$, $\tau \in \{ \tau_1, \tau_2, \cdots\}$ vectorul semnalelor de intrare care se aplică în momentele de timp τ. Evoluția modelului este determinată de uplul $(\mathbf{u}, \sigma)$, notând aceasta prin $S_0[(\mathbf{u}, \sigma * t) > S$ unde t este executată ultima tranziție.

Viabilitatea (engl.: liveness)

O tranziție t este *cavsi-viabila* (engl.: quasi-live) dacă există o pereche $(\boldsymbol{u}, \sigma)$ astfel încât $S_0[(\mathbf{u}, \sigma * t) >$ care până la urmă execută tranziția t.

O tranziție t este *viabilă* (engl.: live) dacă din oricare starea inițială S_0, există o pereche $(\boldsymbol{u}, \sigma)$ astfel încât $S_0[(\mathbf{u}, \sigma * t) >$ adică până la urmă execută tranziția t.

Este relevant de analizat dacă o tranziție este *mereu viabilă* sau numai o singură dată este executabilă.

O rețea marcată (OETPN, M_0) este cvasi-viabilă (quasi-live), respectiv viabilă (live), dacă oricare tranziție a ei este cvasi-viabilă, respectiv viabilă.

Reversibilitatea (engl.: reversibility)

Se referă la posibilitatea sistemului de a reveni din starea curentă (marcajul curent) într-o stare anterioară (marcaj anterior).

Formal: $M \epsilon R(OETPN, M_0)$

$M_0 \epsilon R(OETPN, M)$ sau $S_0 \epsilon R(OETPN, S)$

Această înseamnă că există o pereche $(\boldsymbol{u}, \sigma)$ care duce sistemul din starea curentă într-o stare anterioară.

Un sistem reversibil se poate se poate reinițializa.

Observabilitatea (engl.: observability)

În teoria sistemelor se definește un sistem ca fiind *observabil* dacă se poate determina starea lui curentă prin inferențe din informațiile furnizate la ieșire.

Observabilitatea unui model OETPN este deseori mai complexă, din acest motiv se simplifică acest concept prin referire la un task.

Un task al unui model OETPN este *observabil* dacă se poate infera (determina) starea lui curentă din observațiile informațiilor furnizate la ieșire.

Taskul se va numi *parțial observabil* dacă numai o parte a variabilelor de stare pot fi determinate din valorile furnizate la ieșire.

Observabilitatea este utilă în testarea programelor. Dacă taskurile sunt observabile, testele sunt mai ușor de determinat și sunt mai concludente.

4. Exemple de modele OETPN

Modele ierarhizate

Construirea unui submodel PN_2 în locația p2 al părintelui PN_1.

Fig. 11. Crearea unei subrețele

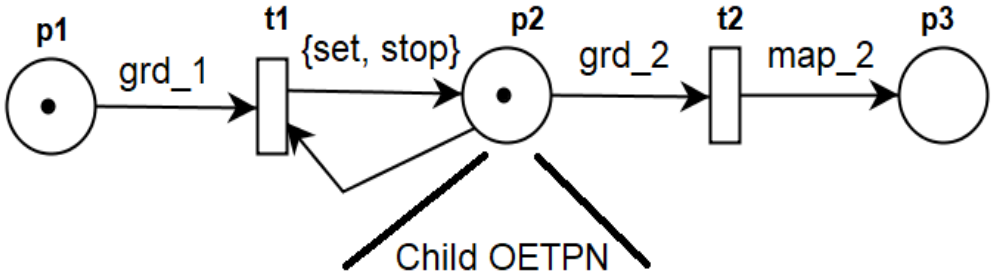
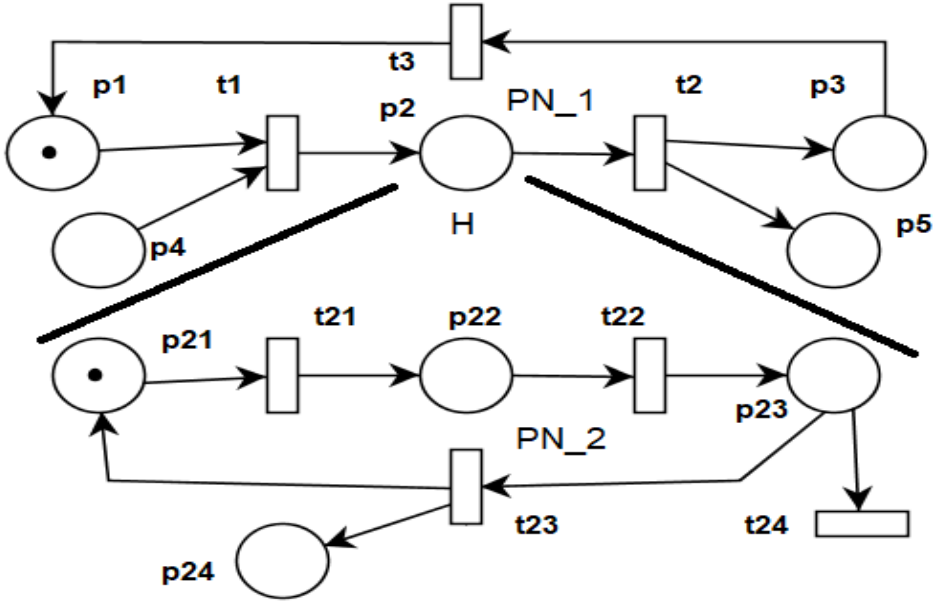


Fig. 12. Exemplu de rețele părinte-copil

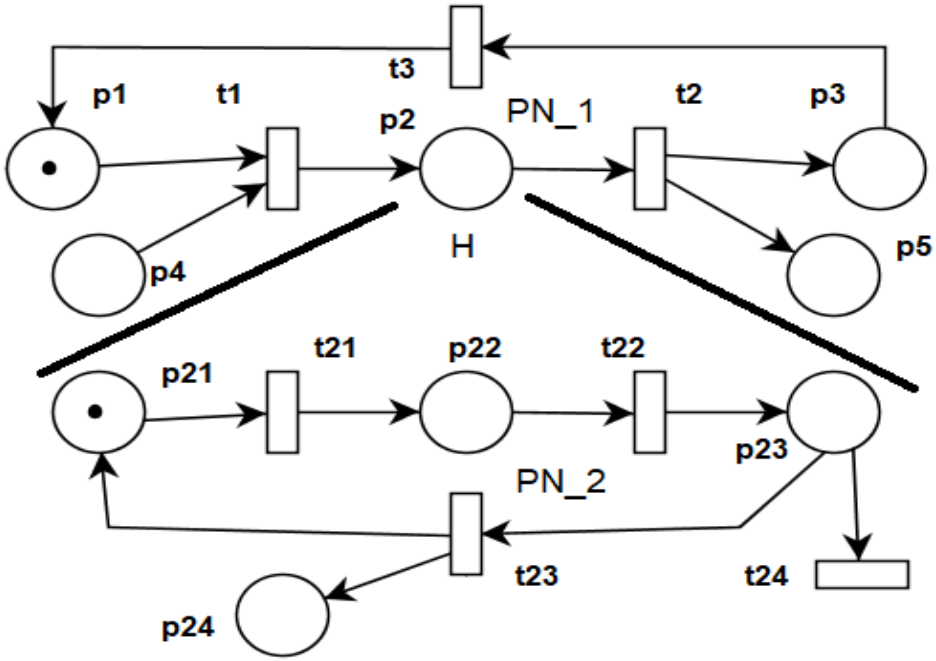
Model OETPN integrează două submodele PN_1 și PN_2. PN_1 este părintele creator al copilului PN_2.

Părintele poate fi creat (instanțiat) și inițializat de către metoda *main()* a programului. Acesta inițializează PN_1 setând un jeton în locația p1.



Se specifică următoarele:

- type(p4) = { parameters}
- type(p2) = PN_2



Tranziția t1 așteaptă sosirea informațiilor în p4 când p1 este nenull, crează folosind mappingul map1 taskul PN_2 în p2, îl inițializează și prin urmare acesta este startat.

Tranziția t2 nu se poate executa (având nevoie de un jeton în p2), până când PN_2 este terminat prin metoda *end*. Aceasta oprește taskul și setează valorile care pot fi folosite de garda și mappingul lui t2.

În varianta 2 canalul de intrare p4 primește un task de tip PN_2, dar nu-l startează (în p4) ci îl setează prin intermediul mappingului lui t1, îl inițializează și apoi îl startează. În acest caz specificațiile locațiilor sunt type(p4)=type(p2)=PN_2. În timp ce în p4 este pasiv, cel creat în p2 este activat imediat după instanțiere.

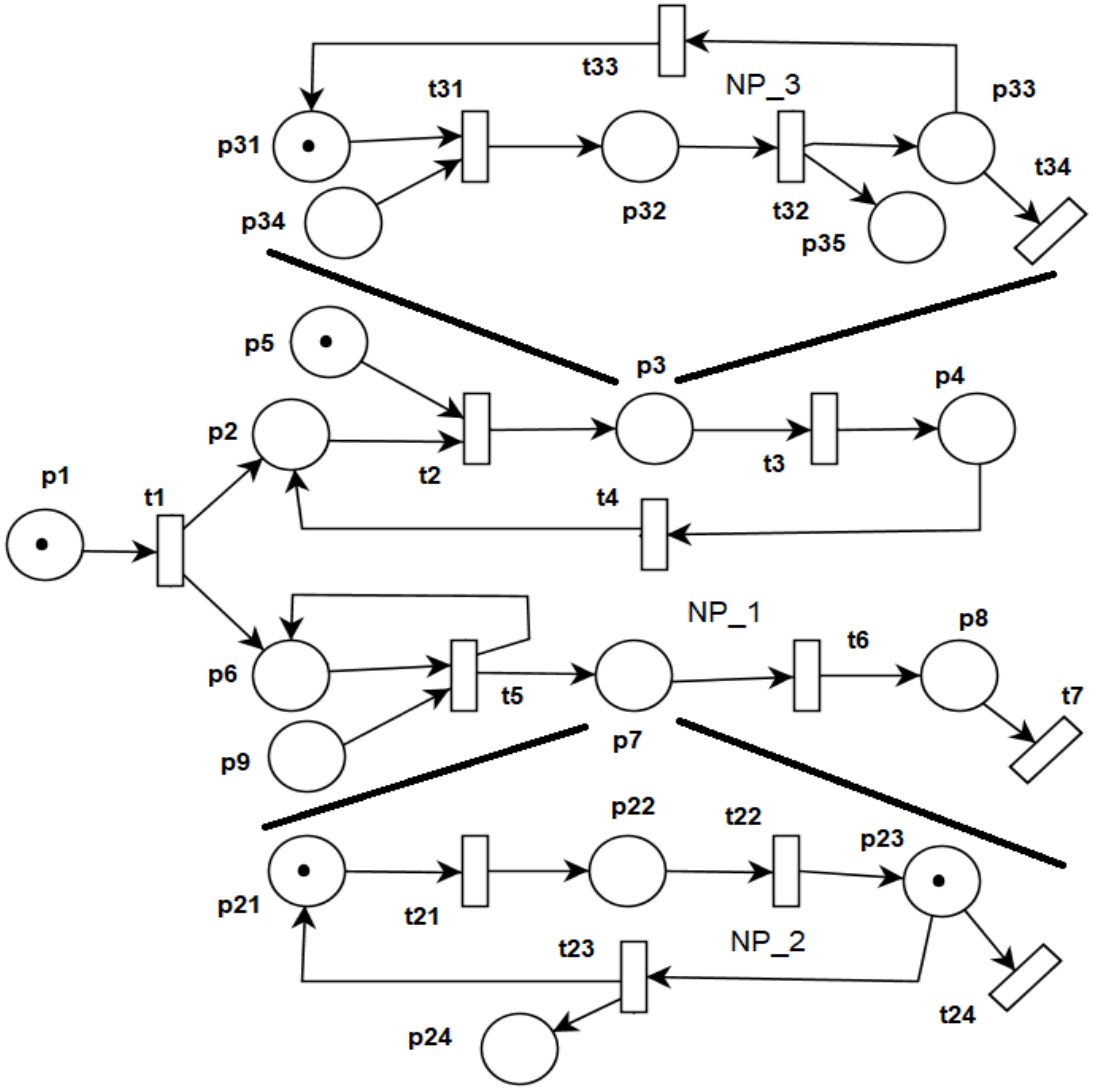
Execuție concurentă

Fig. 13. Modele OETPN copii cu execuție concurentă

Taskuri NP_1, NP_2 și NP_3 care se execută concurent. NP_1 este părintele creator al taskurilor NP_2 și NP_3.

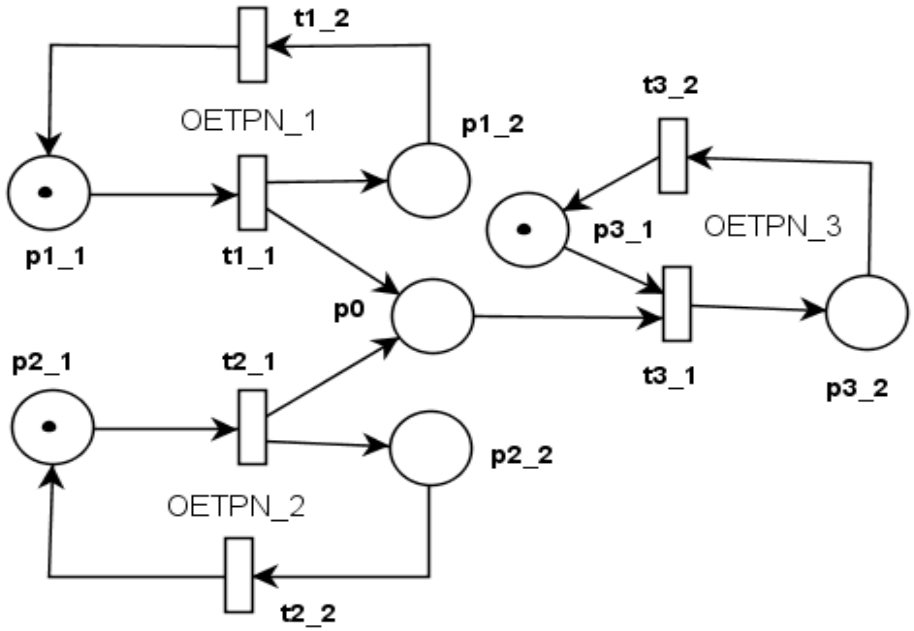
Cei doi copii se execută concurent. Ei pot comunica numai prin intermediul canalelor lor intrare/ieșire.

Încălcarea acestui principiu face imposibilă verificarea programului și poate duce la erori de execuție.



Partiționarea

Fig. 14. Partiționare OETPN cu afectarea ieșirilor



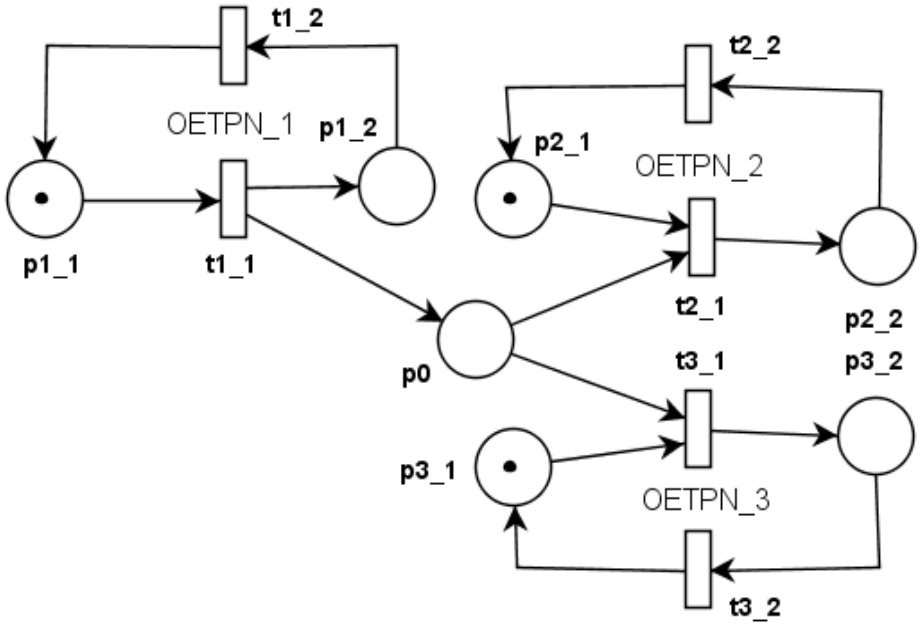
Submodelele OETPN_2 și OETPN_3 extrag jetonul depus în locația p0 de către OETPN_1.

Submodelele au o locație (un canal) de intrare comună.

Executorul fiecăruia „vede” jetonul din p0 ia decizia de a executa tranziția corespunzătoare, ceea ce este greșit deoarece numai un jeton se poate extrage (de către un fir), iar celălalt fir execută o tranziție fără a avea acest drept.

Fig. 15. Partiționarea OETPN cu afectarea intrărilor

Din nou lipsa de coordonare poate duce la anomalii cu efecte dazastruoase.



Într-o rețea Petri clasică este aceasta este un conflict nerezolvabil fără modificarea structurii. În OETPN problema se poate rezolva prin setarea în p0 a unui jeton cu „adresă” care să ientifice destinatarul informației.

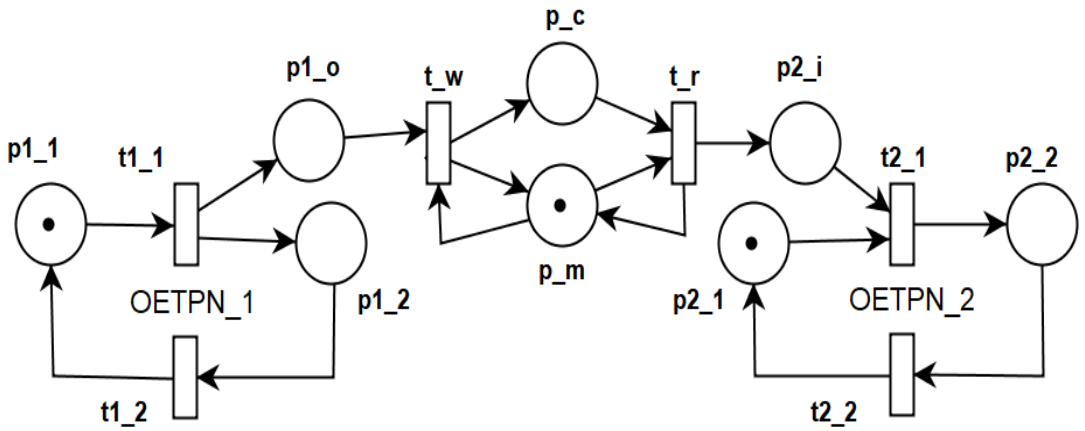
Dacă există o valoare x în $Domain(m(p0))$ astfel încât $grd_{2\_1}(x) = grd_{3\_1}(m(p0)) = true$

➔ t2_1 și t3_1 sunt în conflict și este posibil ca ambele să fie executate simultan.

Dacă se verifică iar condiția precedentă nu este îndeplinită, atunci nu există o anomalie care să se poată manifesta.

Comunicarea a două rețele (modele) OETPN

Fig. 16. Exemplu de comunicare între două modele OETPN



Două rețele OETPN pot comunica prin intermediul unei structuri descrise de locațiile *p_c* și *p_m* folosind tranzițiile *t_w* și *t_r*.

Tipurile locațiilor p1_o, p_c și p2_i trebuie să fie aceleași pentru a putea avea loc schimbul de informații dintre cele două rețele.

Taskul OETPN_1 trimite un jeton prin intermediul canalului său de ieșire p1_0.

Tranziția tw gestionată din afara taskurilor OETPN_1 și OETPN_2 transmite jetonul în p_c.

Blocul t-w*t_r depune jetonul mai departe p2_i.

Împunând gărzi diferite pentru t_w și t_r se pot realiza comunicări între cele două taskuri cu saufără sincronizare.

Excluderea reciprocă și cozi de așteptare

Semaforul este o construcție care poate realiza excluderrea mutuală între mai multe taskuri rezolvând și problema așteptării la coadă FIFO (first-in-first-out) sau non-FIFO.

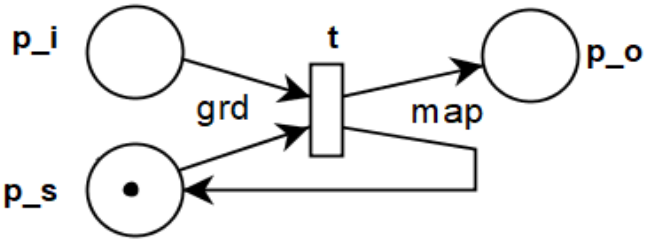
Semaforul este creat ca un obiect OETPN de către un părinte (eventual metoda *main()*) și este utilizat de către modele OETPN pentru sincronizare folosind canalele de intrare/ieșire. Semaforul este un obiect pasiv (pasive block) care are metodele *init*, *request* și *release* pentru inițializarea lui (setarea valorii inițiale), obținerea accesului și respectiv renunțare la acces.

Un OETPN poate folosi pentru intrare într-o regiune critică metoda *request*, iar la ieșire din aceasta metoda *release*. Similar se poate realiza achiziționarea și renunțarea la resurse.

Semaforul poate fi instanțiat cu clasa *Semaphore* (din Java, de exemplu). Modelul OETPN interacționează cu semaforul prin intermediul canalelor sale de intrare/ieșire. Obiectul de tip Semaphore crează lista (coada) de așteptare pe care o gestionează împreună cu metodele de gestionare.

Presupuneri:

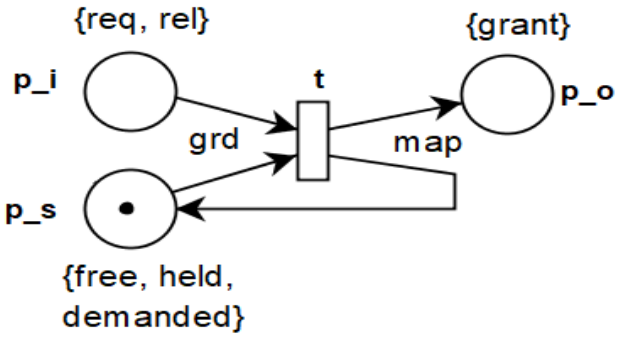
- Fiecare model OETPN este implementat de către un executor (fir de execuție)
- Fiecare *guard* și *mapping* are nevoie de un timp relativ mic pentru execuție (acceptat ca fiind ignorabil)
- Dacă execuția unei metode mapping necesită o durată de timp relativ mare, aceasta este implementată ca un nou fir (adică, executor) corespunzând la un copil OETPN.

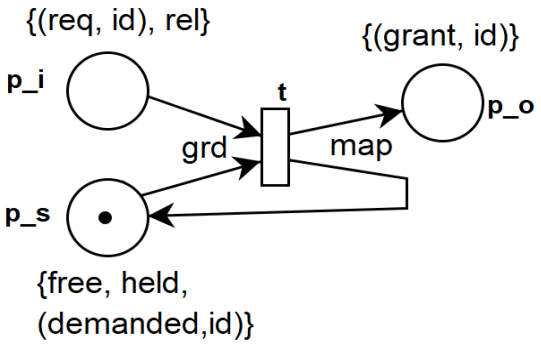


Mutex Class Exemple de excluderi mutuale

Aceasta este o clasă folosită pentru sincronizarea a două taskuri.

- req: request;
- id: requester identifier;
- rel: release;
- grant: signal; idem id;
- free: the semaphore state is free;





- held: the semaphore is granted to an OETPN;
- demanded: the semaphore is requested, and it will be granted when an OETPN task releases it.
- p_i: input channel place
- p_o: output channel place
- p_s: mutex state place

type(p_i) = type(p_s) = type(p_o) = String;
Relațiile de evoluție sunt:
grd^1:= (p_i.value= (req,id) & p_s.value= free);
map^1:= (p_o.value = (grant,id); p_s.value=held);

grd^2 := p_i.value= rel & p_s.value= (demanded,id);
map^2:=p_o.value = (grant,id); p_s.value=held;

grd^3:= p_i.value= (req,id) & p_s.value= held;
map^3:= p_o.value = φ; p_s.value= (demanded,id);

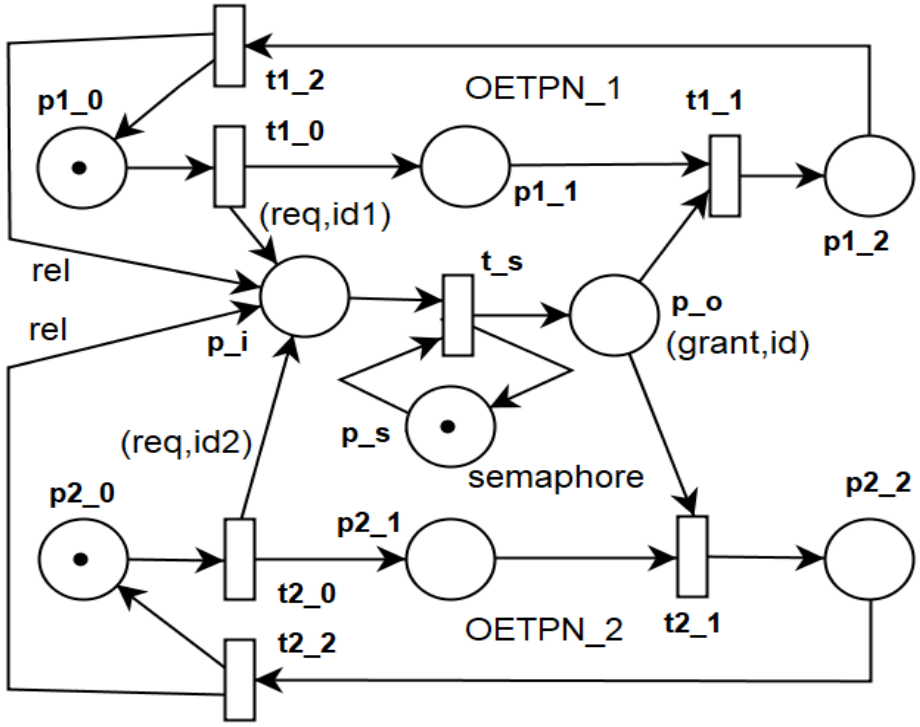
grd^4:= p_i.value= rel & p_s.value= held;
map^4:= p_o.value = φ; p_s.value=free;

Fig. 18. Sincronizarea a două taskuri

Sincronizarea a două modele OETPNs

Sincronizarea modelelor OETPN_1 și OETPN_2 pentru intrarea în regiunile lor critice reprezentate de locațiile p1_2 și respectiv p2_2.

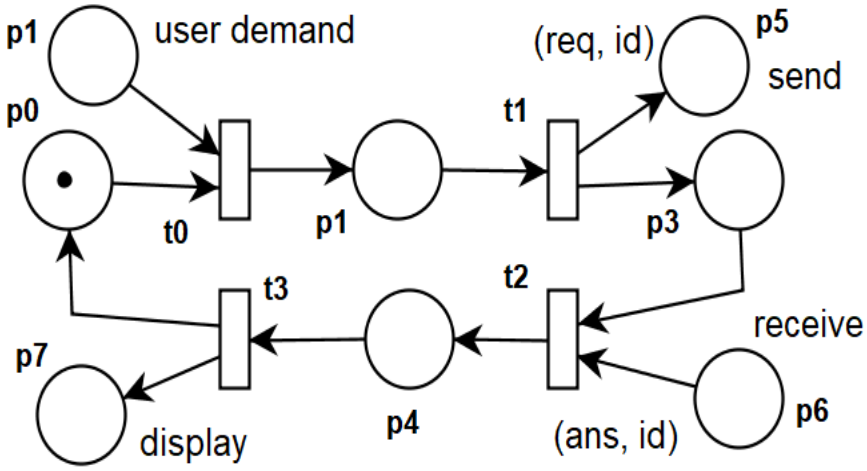
Fiecare task cere acceptarea intrării prin postarea jetonului (req, id1) sau (req, id2) inserat în locația p_i și așteaptă primirea acceptului prin jetonul din locația p_o care va conține și identificatorul beneficiarului.



La ieșirea din secțiunea critică, fiecare task inserează un jeton cu mesajul *rel*. Setarea identificatorului taskului care eliberează resursa nu este relevantă aici.

Model OETPN lucrând cu o bază de date (DB)

Fig. 19. Solicitantul pentru accesarea bazei de date



Prin locația p1 se pot primi solicitări de la un utilizator. Solicitarea este trimisă ca un jeton (req, id) precizând cererea și dentificatorul solicitantului prin canalul de ieșire p5.

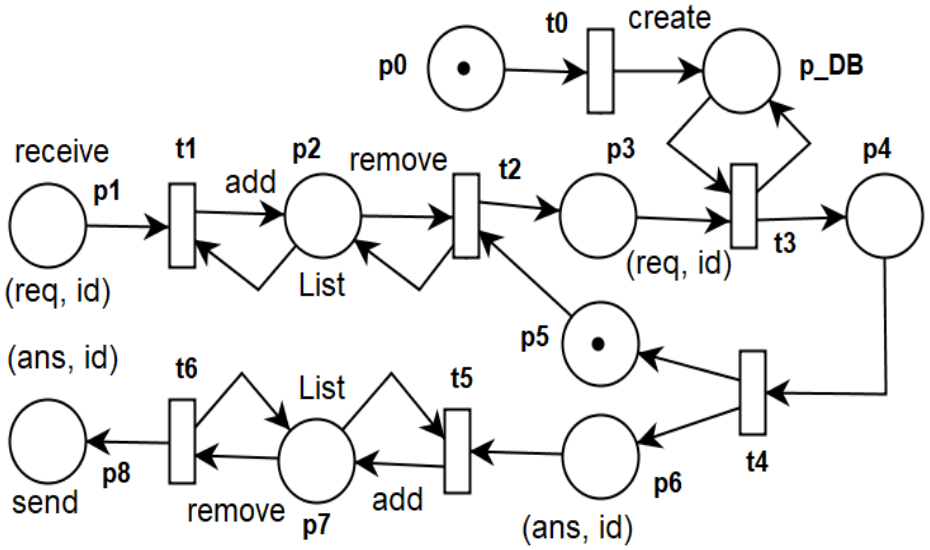
Locația p_6 este un canal de intrare prin care se primește răspunsul de la serverul care gestionează baza de date. Locația p7 este folosită pentru afișarea răspunsului, sau pentru trimiterea lui prin canalul de ieșire la o destinație care este conectată la acesta.

Un utilizator folosește aceste canale pentru a adăuga, elimina, primi sau updata informații în/din baza de date.

Aceleași canale de intrare pot fi folosite pentru ieșirea din conexiune.

Example 2. Server OETPN managing a data bases (DB)

Fig. 20. Serverul de conexiune cu DB



Baza de date este creată de traanziția t0 în locația p_DB prin mappingul ei.\ Structura bazei de date este specificată prin jetonul inițial din p0 sau, dacă p0 este un canal de intrare, prin jetonul depus acolo.

Tranziția t1 servește la primirea cererilor pentru accesare bazei de date prin canalul de intrare p1. O cerere conține perechea *(req, id)* cu *req* mesajul de cereere, iar *id* identificatorul solicitantului. Tranziția t1 adaugă cererea într-o listă stocată în locația p2.

Tranziția t2 preia căte o cerere eliminând-o din listă și o setează în p3, de unde o preia t3 care efectuează solicitarea și depune răspunsul (ans) în p4. Tranziția t4 pune răspunsul în p6 și activează p5 pentru servirea altei solicitări. Mai departe, răspunsul cu identificatorul solicitantului ajunge într-o listă creată în locatiă p7. Prin canalul p8 răspunsul este trimis solicitantului.

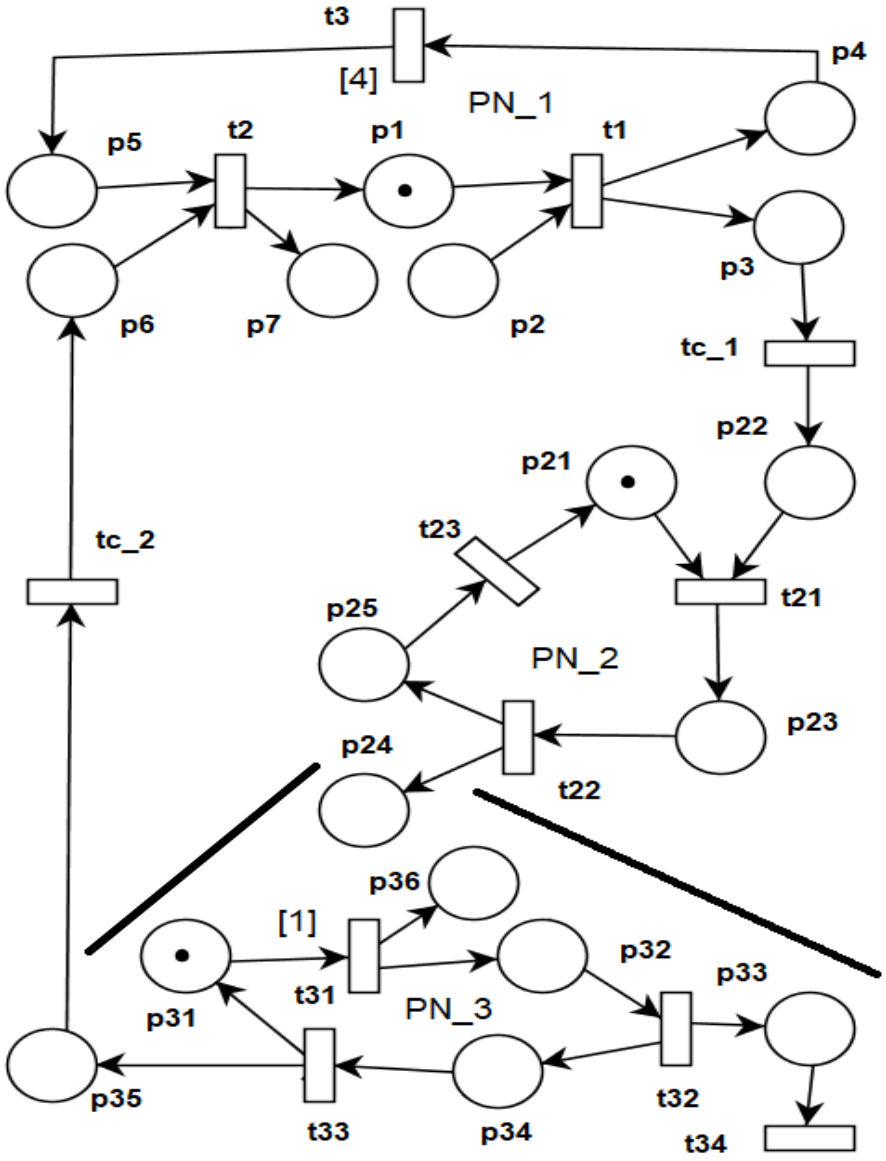
Serverul poate procesa cereri la viteza lui, iar dacă numărul cererilor depășesc viteza lui de procesare, listele (bufferele) din p2 și p7 compensează diferențele temporare de viteză.

Clienții (modelați în Fig. 16) emit solicitări prin p5 (conectat la server în locația p1) și primesc răspuns
de la server prin locația p8 conectată cu locația p6 a clientului.

Fig. 21. Modele distribuite

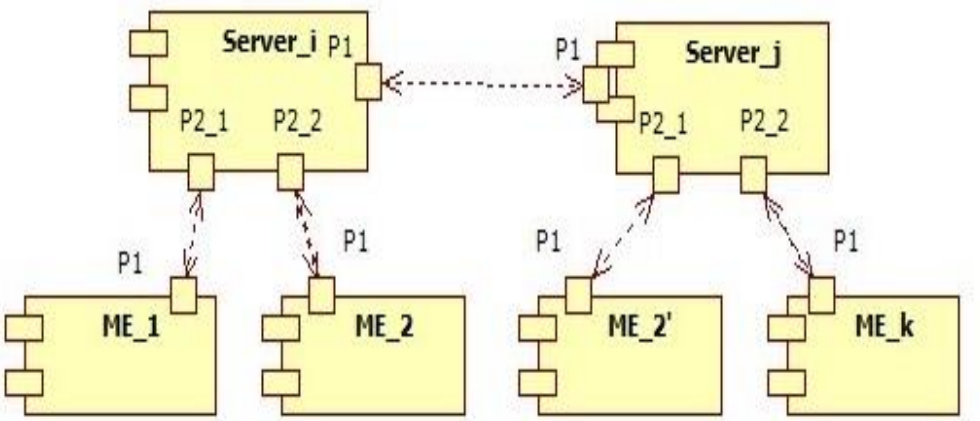
Modele distribuite, transmiterea taskurilor și prelucrări distribuite

Modelul din figura alăturată conține două taskuri implementate pe calculatoare diferite care comunică prin intermediul canalelor de comunicație tc_1 și tc_2. Pentru ca PN_1 să poată transmite taskului PN_2 taskul PN_3 ca să-l execute, este nevoie ca locațiile p3, p22, p23 și p24 să fie de tip PN_3. Pentru ca taskurile să poată comunica între ele trebuie să fie legate la canale de comunicație (de exemplu prin protocolul TCP/IP).



Monitorizarea entităților migratoare (Moving entities monitoring)

Fig. 22. Diagrama de componente



Se consideră un set de entități migratoare (ME) notate cu $\mu_{i,k}$ (i=1,2, …; k=1,2, …) asignate (pentru prima dată) la o zonă Z_i (i=1,2, …) unde au fost create la un server de monitorizare S_i (i=1,2, …).

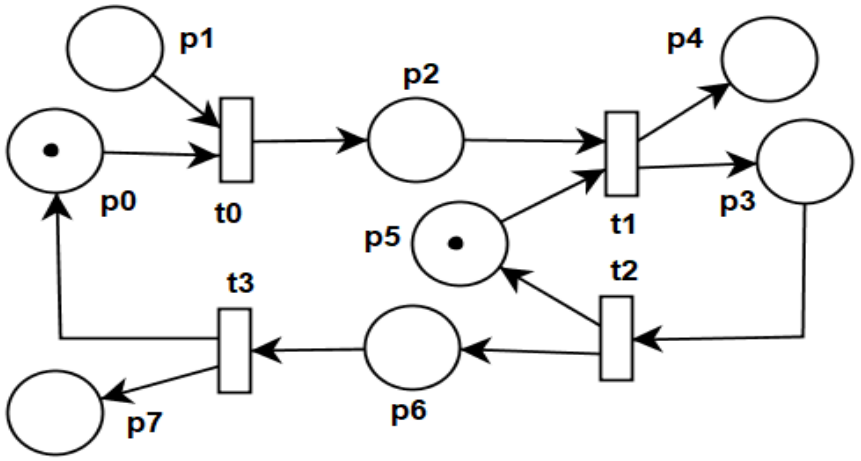
Entitățile $\mu_{i,k}$ pot să se deplaseze liber în oricare zonă Z_j (j=1,2,…).

Când $\mu_{i,k}$ este detectată într-o zonă Z_j ($j \neq i$), serverul S_j este notificat iar el semnalează mai departe serverul S_i unde $\mu_{i,k}$ a fost înregistrată.

Entitățile $\mu_{i,k}$ pot produce asincron evenimente (notate cu $e_{i,k,x}$) care modifică starea lor curentă (notată cu $\sigma_{i,k}$) și cre este semnalată serverului zonei în care se află în momentul respectiv.

Un set C_k (k=1,2, …) de clienți pot să interogheze asincron serveri S_i unde $\mu_{i,k}$ este asignată despre starea curentă a unei entități.

Fig. 23. Modelul OETPN al unei ME



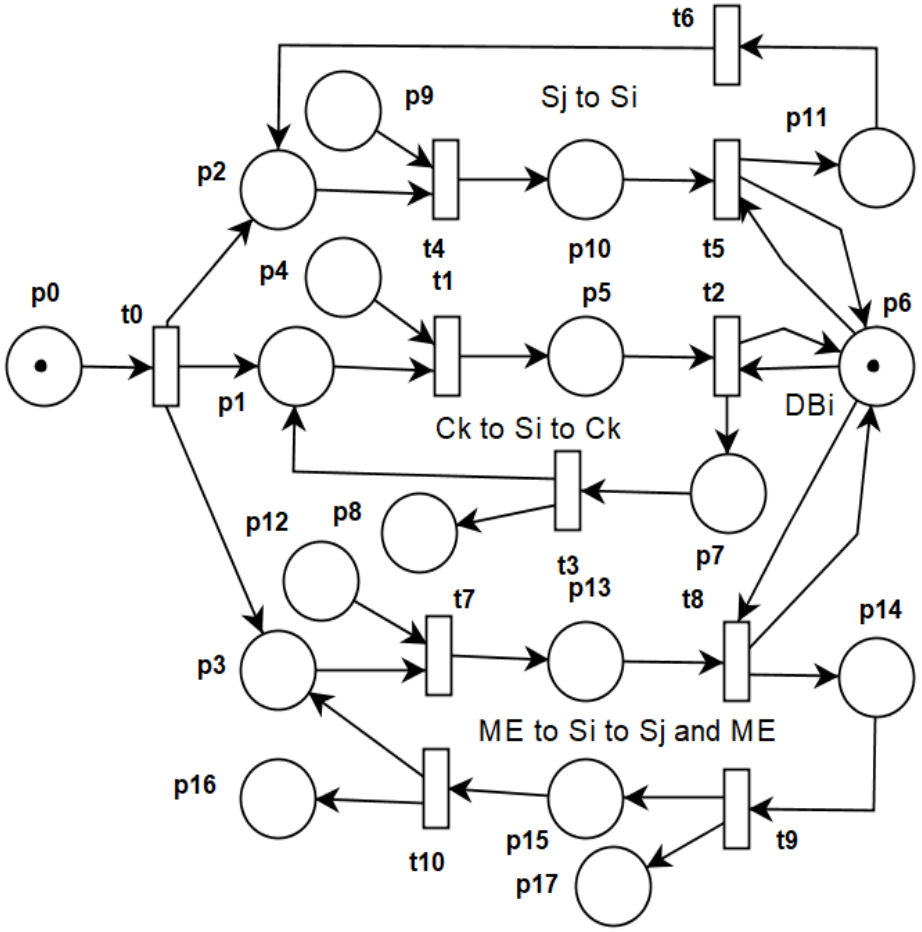
Când ME se mută sau este mutată (deci i se modifică starea), ea generează automat evenimente cu informații pe care le transmite la serverul zonei în care se află.
ME are un canal de intrare (p1) prin care primește informații referitoare la identificatorul zonei în care se află.

Locația p5 conține informații despre identificatorul ei, istoria (icluzând serverul unde a fost înregistrată) și starea ei curentă.

Prin canalul p4 ME trimite serverului local identificatorul, elemente ale istoriei sale și starea curentă. Alte informații pot fi transmise serverului în care a fost înregistrată.

Fig. 24. Modelul OETPN al unui server Si

Modelul unui server Si (care gestionează zona Zi)



- *Sj to Si* prin care serverul Si primește informații prin canalul p9 de la un server Sj și pe care le stochează în baza de date DBi stocată în locația p6.
- *Ck to Si to Ck* prin care serverul Si comunică cu un client Ck. Taskul primește informații prin canalul p4, consultă baza de date DBi și îi răspunde clientului prin canalul p8.
- *ME to Si to Sj and ME* prin care Si este contactat de către o ME prin canalul p12 de la care primește identificatorul și informații de stare. Sunt stocate informațiile primite în DBi. Serverul răspunde entității ME prin canalul p17 și

transmite informații (despre evenimentul curent) serverului unde a fost înregistrată ME (dacă este străină) prin canalul p16.

*

***

END

***

*