Open Access Library Journal
2025, Volume 12, e12969
ISSN Online: 2333-9721
ISSN Print: 2333-9705

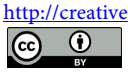
Petri Nets Representation Approach through Chained Linear Lists

Emile Niyongabo[1], Jérémie Ndikumagenge[2], Zihindula Mushengezi Elie[3], Manirabona Audace[2], Longin Ndayisaba[2]

[1]School of Doctoral, University of Burundi, Bujumbura, Burundi
[2]Center for Research in Infrastructure, Environment and Technologies "CRIET", University of Burundi, Bujumbura, Burundi
[3]Institut Supérieur Pédagogique de Bukavu, Bukavu, Democratic Republic of the Congo
Email: niyemi2014@gmail.com, jeremie.ndikumagenge@ub.edu.bi, eliezihindula@yahoo.fr, audace.manirabona@ub.edu.bi, longin.ndayisaba2003@ub.edu.bi

How to cite this paper: Niyongabo, E., Ndikumagenge, J., Elie, Z.M., Audace, M. and Ndayisaba, L. (2025) Petri Nets Representation Approach through Chained Linear Lists. *Open Access Library Journal*, **12**: e12969.
https://doi.org/10.4236/oalib.1112969

Received: January 15, 2025
Accepted: May 26, 2025
Published: May 29, 2025



Abstract

The development of automated systems that meet the desired reliability and efficiency criteria requires the use of scientifically proven modelling tools. Petri nets, which were invented in the 1960s, have allowed designers and analysts to conduct formal studies of future system properties during the modelling stage. Modelling, construction, and synthesis of systems are carried out using various analysis methods derived from the ontological foundation of Petri nets, such as places, transitions, tokens, cover graphs, marker graphs, place invariants, and transition invariants. Each of these methods has benefits and drawbacks. Several models for designing automated systems now exist and coexist, including Markov chains, queues, stacks, graphs, forests and least squares. The current study suggests a novel way to describe Petri nets as chained linear lists. It seeks to give designers and engineers the opportunity to change the initial structure of Petri Nets using processing methods similar to those applied to linear list data structures. As a result, the strategy or technique allows for the broadening of the ontological basis of Petri nets by specifying the requirements for a Petri net representation in the form of chained linear lists.

Subject Areas

Information Management

Keywords

Petri Nets, Chained Linear Lists, Models, Methods, Algorithm

1. Introduction

Engineers employ a variety of methodologies, methods, and strategies for modeling

discrete event automated computing systems [1]. Petri nets provide a variety of ways to design and synthesize varied automated systems that meet the necessary functional requirements. A bipartite graph representation serves as the foundation for petri net modeling [2]. Such a representation allows for the study and detection of the future system's behavioral traits or properties, but does not allow for the addition or removal of any of the network's constituent elements.

Furthermore, translating a Petri net into an ordinary classical graph makes no significant contribution because the reverse operation does not yield a single Petri net from an Euler graph [2].

As a result, the inability to change the structure of a Petri net represented as a bipartite graph eliminates the potential of functional and/or behavioral restructuring and readjustment of the system being studied and designed.

For decades, most automated discrete event systems have been represented with Petri nets. Indeed, since its inception, the Petri net formalism has continued to expand and affect the modeling industries of systems that are diverse in type and shape. Furthermore, this evolutionary process gave rise to the widely used generalized Petri nets [3]. However, the ontological basis of a Petri net is mainly constituted by the concept of state materialized by the number and content of places; the concept of consumable and producible resources materialized by tokens; the passage from one state to another materialized by the activation of one or more transitions; and a range of elements for characterizing them, an abstract framework is defined for modeling, analyzing, and creating systems that match the desired functional criteria using various corrective optimization approaches, models, principles, and techniques [2].

The current Petri net representation models also suffer from restructure issues because it is nearly hard to change the model's functional structure by inserting and/or deleting parts. Furthermore, the absence of iterative methods for building the Petri net model, as well as the absence of processes for tracing the recurrence of indices in a Petri net, demonstrates a lack of flexibility in the design and synthesis of models of discrete event automated systems.

This paper provides a novel way to represent Petri networks as chained linear lists. This type of visual and structural schematization tries to tackle a key problem: the inability to modify the Petri net structure observed in the current representation. As a result, algorithms for manipulating chained linear lists can be applied to them, allowing them to benefit from the advantages associated with linear list processing methods. The design stage allows for changes to the future system's structure and behavior.

2. Material, Tools, Equipment and Methods

2.1. Tools

The work in this paper is based on a generalized Petri net and chained linear lists. Processing models and methods used for linear lists will be extended to generalized

Petri nets.

2.2. Materials

Based on the anonymous system shown in Figure 1 on page 5, we plan to create an algorithm that depicts Petri nets as orthogonal chained linear lists. This model offers a possibility of system initial structure modification, allowing changes to the initial structure of the Petri net by inserting (adding) and/or deleting (eliminating) constituent elements. As the expected automated system' functionalities are crucial for its future usability.

2.3. Methods

The design of the new model for representing generalized Petri nets using chained linear lists must be founded on and articulated around the conservation of the main features. In our current effort, we aim to maintain the existing formalisms provided by the tools and methodologies. The article presents a collection of methods for modifying the functional structure of a Petri net, and therefore the future system. Systems of matrix equations and linear algebras will be used as methods and materials for research, study, and analysis, just as Petri net models in the form of bipolar/bipartite Euler graphs were.

3. Results

3.1. Algorithm for Representing Petri Nets as Chained Linear Lists

In order to elaborate and build an iterative algorithm that allows the encoding of Petri nets in the form of two-dimensional chained linear lists, we shall act on the incidence matrix of an anonymous Petri net shown in Figure 1.

3.2. Structured Types of the Algorithm

The algorithm has two structured types. One for the places recording, the other for the transitions recording.

Component of the "place record" is given on the sheet below:

```
defType PList
        Number place;
        Number weight;
        PList ↑pNext;
          End
```

- **Number place**: this variable indicates the number places.
- **Number weight**: is an element of the incidence matrix. It is given by the difference between the arc rating of an arrow leaving the transition to the position and the arc rating of an arrow leaving the same position to the same transition. If there is no arrow between the transition and the position and vice versa, the arc rating is zero.
- **PList ↑pNext**: represents a pointer to the record of a subsequent place.

Component of the 'transition record' is given on the sheet below:

```
defType TList
                Number transition;
                PList ↑pEl;
                TList ↑tNext;
        End
```

- **Number transition**: This variable indicates the number of transitions;
- **PList ↑pEl**: represents a pointer to the place record;
- **TList ↑tNext**: represents a pointer to the next transition record.

Listing 1 below gives the set of instructions sequence for a function that calculates an element of the incidence matrix.

```
1  Function CalculatingWeight (Number i, Number j)
2
3      Number weight;
4
5      Begin
6
7          if(Arc from Pi to Tj does not exist)
8              bow_weight(Pi, Tj)=0 ;
9          Endif
10
11         if(Arc from Tj to Pi does not exist)
12             bow_weight(Tj, Pi)=0 ;
13         Endif
14
15         weight= bow_weight(Tj, Pi) - bow_weight(Pi, Tj) ;
16
17 return weight;
```

P_i : is the i^{th} place.

T_j : is the j^{th} transition.

$weight$: is the arc of the arrow.

$bow\_weight\left(P_i, T_j\right)$: This is the evaluation of the arc that connects the i^{th} place to the j^{th} transition.

$bow\_weight\left(T_j, P_i\right)$: This is the evaluation of the arc that connects the j^{th} transition to the i^{th} place.

Listing 2 below gives the sequence of instructions for an algorithm representing Petri nets in the form of chained linear lists.

```
1. Algorithm RdP_StructuresLinearList
2.      defType     PList
3.                              PList ↑pNext;
4.                              Number weight;
5.                              Number place;
6.      End
7.
8.      defType TList
9.                          TList ↑tNext ;
10.                       PList ↑pEl ;
11.                       Number transition ;
12.     End
13.
```

```
14.  TList ↑thead ;
15.  TList ↑tC ;
16.  Number i, j ; // indexes
17.  Number m ; // Number of transitions
18.  Number n ; // Number of places
19.  PList↑pC[m];// show the number of queues for each list of places
20.
21.  debut
22.  i←1 ;
23.  j←1 ;
24.  Repeat set(m) ; until (m>0) ;
25.  Repeat set(n) ; until (n>0) ;
26.  thead ← new TList ;
27.  thead↑.transition←1 ;
28.  tC ←thead ;
29.  tC↑.tNext ← nil ;
30.
31.  While (j<=m)
32.          if(j>1)
33.                  tC↑.tNext ← new TList ;
34.                  tC← tC↑.tNext ;
35.                  tC↑.transition←j;
36.                  tC↑.tNext ← nil ;
37.          EndIf
38.
39.          tC↑.pEl↑.pNext ← nil ;
40.          tC↑.pEl↑.pNext ← new PList ;
41.          pC[j] ← tC↑.pEl↑.pNext;
42.          pC[j]↑.place ← 1;
43.          pC[j]↑.weight ← CalculatingWeight(1,    j) ;
44.          pC[j]↑.pNext ← nil ;
45.
46.          i←2 ;
47.          While (i<=n)
48.                  pC[j]↑.pNext ← new PList ;
49.                  pC[j] ← pC[j]↑.pNext;
50.                  pC[j]↑.place←i ;
51.                  pC[j]↑.weight←CalculatingMasse(i, j) ;
52.                  pC[j]↑.pNext←nil ;
53.                  i++ ;
54.          EndWhileI
55.          j++ ;
56.  EndWhileJ
```

57. **End Algorithm**

4.3. Components of the Algorithm

Let's consider the declaration of variables bellow:

- **Number n:** declaration of a variable *n* which indicates the total number of places to be entered from the keyboard.
- **thead↑.pEl↑.pNext ← new PList**: reservation of memory space for a pointer to the place element;
- **Number m**: declaration of a variable *m* which indicates the total number of transitions to be entered from the keyboard.
- **Pitstop[m]**: declaration of a tail list of type **PList**. The variable m denotes the total number of transitions. Each transition is a head for a list of place elements (place column). So the total number of heads for all columns of places (transitions) must be equal to the total number of tails for all columns.
- **TList ↑thead:** declaration for a pointed variable **thead** of type **TList**;
- **thead ← new TList:** reservation of memory space for a variable pointed to **thead** of type **TList**;
- **TList ↑tC**: declaration for a pointed variable **tC** of type **TList**;
- **thead↑.transition ←1:** assignment of the first element of the list. This first element is at the same time the last element of the list at the beginning, which implies that the tail **tC** points to the head **thead**. So **tC ←thead**;
- **tC↑.tNext ← nil:** the successor of the tail of the **tC** transition is **nil**;
- **thead↑.pEl↑.pNext:** a pointer to a place element.

4.4. Case Study of an Anonymous Petri Net

This illustration is an anonymous representation of a Petri net consisting of three transitions and three places. The variables m and n denote the number of transitions and positions, respectively. We used this anonymous scenario, Figure 1, to demonstrate the applicability of the results and confirm the pragmatic element of this novel approach in the tables below, Table 1 and Table 2.

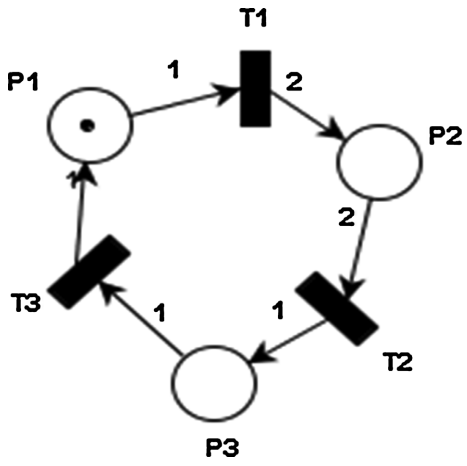


Figure 1. Example of an anonymous Petri net.

Table 1. Verification of the algorithm.

Applying the above algorithm, we have: $i \leftarrow 1, j \leftarrow 1, m \leftarrow 3$ and $n \leftarrow 3$.

Valeur de *j*	Valeur de *i*	Valeur des variables
		thead ← new TList ; thead↑.transition←1 ; tC ← thead ; tC↑.tNext ← nil ;
1	1	tC↑.pEl↑.pNext ←nil ; tC↑.pEl↑.pNext ←new PList ; pC[j] ← tC↑.pEl↑.pNext ; pC[j]↑.place ←1 ; pC[j]↑.weight ← -1 ; pC[j] ↑.pNext ← nil ; i←2 ;
j	2	pC[j]↑.pNext ←new PList ; pC[j] ← pC[j]↑.pNext ; pC[j]↑.place ←2 ; pC[j]↑.weight ←2 ; pC[j]↑.pNext ←nil ; i←3 ;
1	3	pC[j]↑.pNext ←new PList ; pC[j] ← pC[j]↑.pNext ; pC[j]↑. place ←3 ; pC[j]↑.weight ←0 ; pC[j]↑.pNext ←nil ; i←4; j←2;
2	1	tC↑.tNext ← new TList ; tC ← tC↑.tNext ; tC↑.transition←2 ; tC↑.tNext←nil ; tC↑.pEl↑.pNext ←nil ; tC↑.pEl↑.pNext ←new PList ; pC[j] ← tC↑.pEl↑.pNext ; pC[j]↑.place ←1 ; pC[j]↑.weight ← 0 ; pC[j]↑.pNext ← nil ; i←2 ;
2	2	pC[j]↑.pNext ←new PList ; pC[j] ← pC[j]↑.pNext ; pC[j]↑.place ←2 ; pC[j]↑.weight ←-2 ; pC[j]↑.pNext ←nil ; i←3 ;
2	3	pC[j]↑.pNext ←new PList ; pC[j] ← pC[j]↑.pNext ; pC[j]↑.place ←3 ; pC[j]↑.weight ←1 ; pC[j]↑.pNext ←nil ; i←4 ; j←3;

Continued

3	1	tC↑.tNext ← new TList ; tC ← tC↑.tNext ; tC↑.transition←3 ; tC↑.tNext←nil ; tC↑.pEl↑.pNext ←nil ; tC↑.pEl↑.pNext ←new PList ; pC[j] ← tC↑.pEl↑.pNext ; pC[j]↑.place ←1 ; pC[j]↑.weight ← 1 ; pC[j]↑.pNext ← nil ; i←2 ;
3	2	pC[j]↑.pNext ←new PList ; pC[j] ← pC[j]↑.pNext ; pC[j]↑.place ←2 ; pC[j]↑.weight ←0 ; pC[j]↑.pNext ←nil ; i←3 ;
3	3	pC[j]↑.pNext ←new PList ; pC[j] ← pC[j]↑.pNext ; pC[j]↑.place ←3 ; pC[j]↑.weight ←-1 ; pC[j]↑.pNext ←nil ; i←4 ; j←4;

4.5. Modification Operation on a Petri Net Represented as or by a Chained Linear List

Let us demonstrate how to change a Petri net expressed in the form of chained linear lists by adding a member at the end. To keep things simple and reduce workload, we'll focus on putting a transition element into a Petri net. The approach outlined below allows for the addition of transitions and/or places at the end of the list. The formal arguments t and p of type Number specify the number of transitions and places to be added. The values of these variables must be strictly greater than zero, with n and m denoting the number of places and current number of transitions.

```
Procedure AddNodeTransitionPlaceEnd (Number n, Number m,Number p, Number t)
Number j, k,i ;
Begin
    if(t>0 et p==0)
        j←m+1 ;
    EndIf

    if(p>0 et t==0)
        j←1;
    EndIf
```

```
        if(p > 0 et t>0)
          j←1 ;
      EndIf

        While (j ≤ (m+t))
            if(t>0 et j > m)// It is used to add transitions only
                tC↑.tNext ← new TList ;
              tC              ←              tC↑.tNext              ;

                tC↑.tr ← j ;
                tC↑.tNext ← nil ;
                  tC↑.pEl↑.pNext ← nil ;
                  tC↑.pEl↑.pNext ← new PList ;
                  pC[j] ← tC↑.pEl↑.pNext ;
                  pC[j]↑.place ← 1;
                  pC[j]↑.weight ← CalculatingWeight ( 1, j);
                  pC[j]↑.pNext ← nil ;
            EndIf
            if(p==0 et t > 0)
                  i←2 ;
            EndIf
             if(p> 0 et t >0 et j>m)
                 i←2 ;
          EndIf

            if(p> 0 et t==0)
              i←n+1 ;
            Endif
               if(p > 0 et t>et j ≤ m)
             i ← n+1;
           EndIf
             While (i ≤ (n+p))
                      pC[j]↑.pNext ← new PList ;
                      pC[j]← pC[j]↑.pNext;
                      pC[j]↑.place ← i;
                      pC[j]↑.weight ← CalculatingWeight(i, j);
                      pC[j]↑.pNext ← nil;
                      i++;
            EndWhileI
            j++ ;
        EndWhileJ
EndProcedure
```

Based on Figure 1, the insertion of two places and three transitions, imply the calling of procedure **AddNodeTransitionPlaceEnd (2, 2, 2, 3);**

For procedure's accuracy and validation; we perform or run checking processes as shown in Table 1.

Table 2. Procedure for inserting transition or place nodes at the end.

Variables n,m,p,t,i et j	In While loop by j	In While loop by i	Values
$n=2,m=2,$ $p=2,t=1;$			
if $(p>0$ and $t>0)$ $j=1$			
$m+t=2+1>j=1$	$j=1$		
	$j=1$ $j\le(m+t)\Leftrightarrow 1\le(2+1)$		
	$j=1$ $t>0$ and $j>m\Leftrightarrow 1>0$ and $1>2$		
	$j=1$ if $(p>0$ and $t>0$ and $j\le m)$ $i=n+1\Leftrightarrow i=2+1$		

	$j=1$	$i=3$ $i=3\le(n+p)\Leftrightarrow 3\le(2+2)$	`pC[j]↑.pNext ← new PList ;`
			`pC[j]← pC[j]↑.pNext;`
			`pC[j]↑.place ← 3;`
			`pC[j]↑.weight ← CalculatingWeight(3, 1);`
			`pC[j]↑.pNext ← nil;`

	$j=1$	$i=4$ $i=4\le(n+p)\Leftrightarrow 4\le(2+2)$	`pC[j]↑.pNext ← new PList ;`
			`pC[j]← pC[j]↑.pNext;`
			`pC[j]↑.place ← 4;`
			`pC[j]↑.weight ← CalculatingWeight(4, 1);`
			`pC[j]↑.pNext ← nil;`

	$j=2$ $j\le(m+t)\Leftrightarrow 2\le(2+1)$		
	$j=2$ if $(t>0$ and $j>m)\Leftrightarrow 1>0$ and $2>2$		
	$j=2$ if $(p>0$ and $t>0$ and $j\le m)$ $i=n+1\Leftrightarrow i=2+1=3$		

	$j=2$	$i=3$ $i=3\le(n+p)\Leftrightarrow 3\le(2+2)$	`pC[j]↑.pNext ← new PList ;`
			`pC[j]← pC[j]↑.pNext;`
			`pC[j]↑.place ← 3;`
			`pC[j]↑.weight ← CalculatingWeight(3, 2);`
			`pC[j]↑.pNext ← nil;`

	$j=2$	$i=4$ $i=4\le(n+p)\Leftrightarrow 4\le(2+2)$	`pC[j]↑.pNext ← new PList ;`
			`pC[j]← pC[j]↑.pNext;`
			`pC[j]↑.place ← 4;`
			`pC[j]↑.weight ← CalculatingWeight(4, 2);`
			`pC[j]↑.pNext ← nil;`

Continued

$j=3$ $j \le (m+t) \Leftrightarrow 3 \le (2+1)$		

		`tC↑.tNext ← new TList ;`
		`tC← tC↑.tNext;`
		`tC↑.tr ← 3 ;`
		`tC↑.tNext ← nil ;`
		`tC↑.pEl↑.pNext ← nil ;`
		`tC↑.pEl↑.pNext ← new PList ;`
		`pC[j] ← tC↑.pEl↑.pNext ;`
		`pC[j]↑.place ← 1;`
		`pC[j]↑.weight ←`
		`CalculatingWeight(1,3);`
		`pC[j]↑.pNext ← nil ;`

$j=3$ if $(t>0$ and $j>m) \Leftrightarrow 1>0$ and $3>2$		
$j=3$ if $(p>0$ and $t>0$ and $j>m)$ $i \leftarrow 2;$		

$j=3$	$i=2$
	$i=2 \le (n+p)$
	$\Leftrightarrow 2 \le (2+2)$
		`pC[j]↑.pNext ← new PList ;`
		`pC[j]← pC[j]↑.pNext;`
		`pC[j]↑.place ← 2;`
		`pC[j]↑.weight ←`
		`CalculatingWeight(2, 3);`
		`pC[j]↑.pNext ← nil;`

$j=3$	$i=3$
	$i=3 \le (n+p)$
	$\Leftrightarrow 3 \le (2+2)$
		`pC[j]↑.pNext ← new PList ;`
		`pC[j]← pC[j]↑.pNext;`
		`pC[j]↑.place ← 3;`
		`pC[j]↑.weight ←`
		`CalculatingWeight(3, 3);`
		`pC[j]↑.pNext ← nil;`

$j=3$	$i=4$
	$i=4 \le (n+p)$
	$\Leftrightarrow 4 \le (2+2)$
		`pC[j]↑.pNext ← new PList ;`
		`pC[j]← pC[j]↑.pNext;`
		`pC[j]↑.place ← 4;`
		`pC[j]↑.weight ←`
		`CalculatingWeight(4, 3);`
		`pC[j]↑.pNext ← nil;`

5. Results and Discussions

The strategy for encoding Petri nets as chained linear lists involves two structures: one for **PList** placements and one for **TList** transitions. This is the orthogonal representation of chained linear lists, similar to a two-dimensional matrix.

The transition cell pointer points to the following 'transition' record, whereas the place cell pointer points to the next 'place' record. This innovative technique offers numerous advantages. It enables for faster node element insertion and deletion operations than fixed-dimension arrays, which require a lot of RAM. Actually, the allocation of RAM which use linear lists results in low memory space consummation. Furthermore, access to list components is indirect, except the list Head and Tail [4] [5].

It also allows for more freedom when entering and deleting list nodes, as well as displaying list components. As a result, it makes it easier to modify the functional

and behavioral features of the designed system [5].

6. Conclusion

The iterative approach for expressing Petri nets as chained linear lists allows the insertion or deletion of places and/or transitions at the end of a Petri net. This novel approach is spatially efficient and enables the use of list processing algorithms while studying and analyzing Petri nets. In particular, it allows for minor changes to a Petri net, and therefore the functional and behavioral structure of the future system being studied and designed.

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] Piétrac, L. and Denis, B. (1999) Une approche de méta-modélisation formelle des méthodes de conception des systèmes automatisés de production.

[2] Peterson, J.L. (1981) Petri Net Theory and the Modeling of Systems. Prentice Hall PTR.

[3] Boufaden, A., Pietrac, L. and Gabouj, S. (2005) L'usage des réseaux de Petri dans la théorie de contrôle par supervision. *Sciences et Technologies de l'Automatique*, **2**, 10 p.

[4] Mbunga, L.L. and Mayekela, P.M. (2017) Utilisation D'une Methode Multicritere D'aide A La Decision Pour Le Choix D'une Structure De Donnees Dans Un Probleme De Gestion. *International Journal of Innovation and Applied Studies*, **20**, 711-723.

[5] Bouchiha, D. (2020) Algorithmique Et Programmation En Pascal: Cours Avec 190 Exercices Corrigés. Éditions Universitaires Européennes.