Algorithm Modelling with Petri Nets - Comparison with Data Dependence Graphs

Zdeněk Hanzálek

Trnka .... Department of Control Engineering, Karlovo nám. 13
Czech Technical University in Prague, 121 35 Prague 2, Czech Republic
tel: +42 02 24357434, fax: +42 02 24357298, E-mail: hanzalek@rtime.felk.cvut.cz
and LAG INPG Grenoble

May 26, 1998

Abstract

This article focuses on algorithm representation by means of Petri Nets. The basic structural features of algorithms are dictated by their data and control dependencies. These dependencies refer to the precedence relations of computation that need to be satisfied in order to compute the problem correctly. The absence of dependencies indicates the possibility of simultaneous computations (the fewer dependencies the bigger parallelism). This article presents a modeling strategy, which is coherent for noniterative as well as iterative problems. We show that the data dependencies of iterative problems with 'general uniform constraints' can be modelled by an event graph (marked graph). We distinguish between two modeling approaches - the first based on the problem analysis and the second based on the sequential algorithm for which we introduce the term IP-dependencies. We show how to simplify the model by reduction of implicit places and self-loop places. Importance of antidependencies and output dependencies is underlined and algorithm transformations based on Petri Net analysis is shown. This original approach allows us to put a knowledge of automatic parallelization via data dependence graphs and Petri Nets to the same theoretical platform and to join the two scientific branches.

1 Model based on the problem analysis

In this paragraph we will focus on the situation when the model is constructed directly from the problem specification given by n statemnts. It

means that we do not make use of any sequential algorithm, because the sequential algorithm specifies in which order the instructions will be performed. In other words, when we construct a model directly at the moment when we make the problem analysis and there is no conditional branching then the model contains just data dependencies.

The problems with uniform constraints (reference) consist of n statements having the following form:

$$x_i(k) = f_i(x_1(k - \beta_{i1}), ..., x_n(k - \beta_{in})) \tag{1}$$

where k is the iteration index (k =0,1,2,....), and β_{i1} to β_{in} are constants from $\mathcal{N}$.

Definition 1 *The triple (G, L, H) is a uniform graph if it is such that:*
$G(V, E)$ is a directed graph where V is a set of vertices and E is a set of edges
$L : E \rightarrow \{\mathcal{N} + 0\}$ is a length associated to each edge
$H : E \rightarrow \mathcal{N}$ is a height associated to each edge

With the aim to compare Data Dependence Graphs (DDGs) and Petri Nets (PNs) we introduce a new term 'general uniform constraints' modeled by general uniform graph (where $L : E \rightarrow \{\mathcal{Z}\}$ is a length associated to each edge) or special marked graph alowing negative number of tokens.

Modelled iterative problems are supposed to consist of n statements having in general the following form:

$$x_i(k + \alpha_i) = f_i(x_1(k - \beta_{i1}), ..., x_n(k - \beta_{in})) \tag{2}$$

where:
k is the iteration index (k =0,1,2,...,)
α,β are constants from $\mathcal{Z}$ (this assumption implay that also negative number of tokens will be under considaration)
$f_1,...,f_i,...,f_n$ are functions $R^n \longrightarrow R$ represented by the PN transitions
$x_1,...,x_i,...,x_n$ are variables from R represented by the PN places (each variable is represented by so many PN places as many times it is read)

> **Rule 1:** *The PN model of iterative problems consisting of statements given by Equation 2 can be constructed in the following way:*
> *1) create the transition T_i for each statement given by the function f_i with the ouput places corresponding to the output variables*

2

> *(matrix Post) - there are so many output places corresponding*
> *to a given variable as many times the variable is used*
> *2) put α tokens into the transition output places*
> *3) draw arcs from places to transitions (matrix Pre)*
> *4) put β tokens into the transition input places.*

Remark: Notice that if some α and β are negative numbers then the resulting number of tokens need not be negative.

Finally, a positive number of tokens in the PN model corresponds to variables that have to be initialized. An algorithm, reading data before their actualisation, implies that a certain part of the model is not live (e.g. comprising a negative number of tokens in a place).

2   Model based on the sequential algorithm

The way of data representation by the PN places does not match the real function of the data in the computer. The data in the computer memory are written once and could be read several times but the token in the PN is put to the place once and got also once. In other words this way of representing an algorithm by PNs is not exactly correct because PNs were designed to represent flows of control and not flows of data. For a correct use of PNs it is necessary to have the following restriction: each datum is represented by so many places as many times the datum is used.

2.1   Acyclic algorithms

In many sequential programs, there are dependencies that are introduced artificially by the programmer and that may be eliminated. Imagine for example an algorithm for the noniterative problem stated in Figure **??**:

Example 1:

```
T0 : A = X . X
T1 : B = A - 1
T2 : C = A + 1
T3 : Y = C . B
```

Example 1 sequentializes execution of statements T_1 and T_2. This kind of control dependencies is given by an instruction pointer and will be called IP dependencies in the rest of this article. IP dependencies will be modelled

by thick lines. When modelling a sequential algorithm without conditional branching there should always be an oriented path of IP dependencies going through all transitions and holding just one token. It is obvious that IP dependencies are very often implicit to data dependencies. In other words data dependencies order partially execution of statements while IP dependencies order totaly execution of statements.
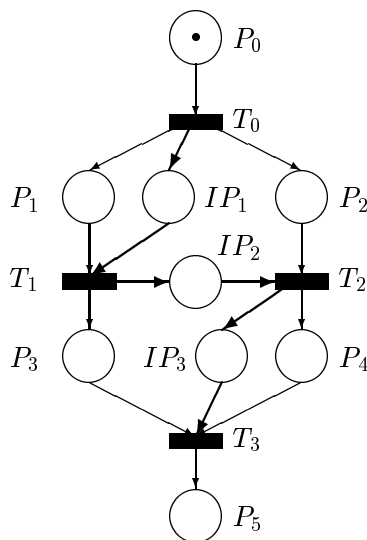
The model of the algorithm given in Example 1 is shown in Figure 1.

Figure 1: Representation of the algorithm from Example 1

2.2   Cyclic algorithms

Example 2: *consider an example algorithm given by the following pseudo-code:*

$k = 0;$
while $TRUE$
$\quad k = k + 1$
$\quad x_1(k) = 3 * x_7(k-1)$
$\quad x_2(k+2) = x_1(k-2) + 4$
$\quad x_3(k) = x_1(k)/6$

$$x_4(k+1) = 2 * x_3(k)$$
$$x_5(k-1) = x_2(k+1) + x_4(k+1)$$
$$x_6(k) = x_3(k-1) + 100$$
$$x_7(k) = x_5(k-1)/x_6(k)$$

endwhile;

This code is represented by a PN model given in Figure 2. The PN model is constructed with use of Rule 1, in addition it includes IP dependencies. Markings correspond to the presence of valid data computed by the previous transition or assigned to in the algorithm initialisation.
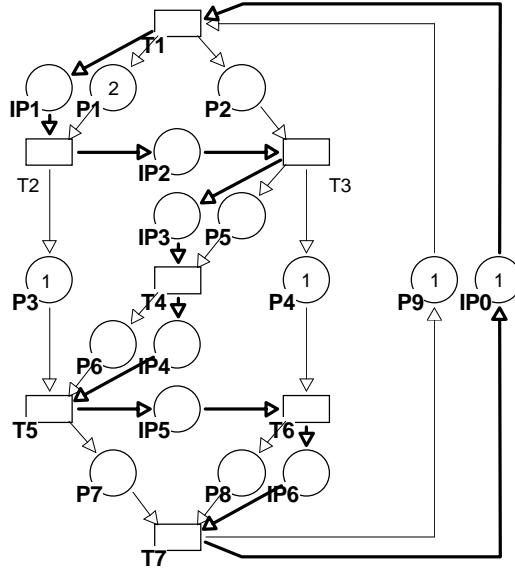

Figure 2: PN representation of Example 2

2.3 Detection and removal of antidependencies and output dependencies in a PN model

Many dependencies are inherent, and cannot be removed. However they may be modified by various transformation techniques. Profound analysis of this problem was done by researchers using data dependence graphs (see [7] for more details). The aim of this article is to show that similar analysis

can be done also with the use of Petri Nets. It is important to notice that the analysis done by DDGs is based on a computer art knowledge. On the other hand the analysis done by means of Petri Nets brings a new perspective arising from the fact that Petri Nets are a mathematical and visualisation tool with profound theoretical background.

Example 3: *consider a simple sequence of statements containing an antidependence:*
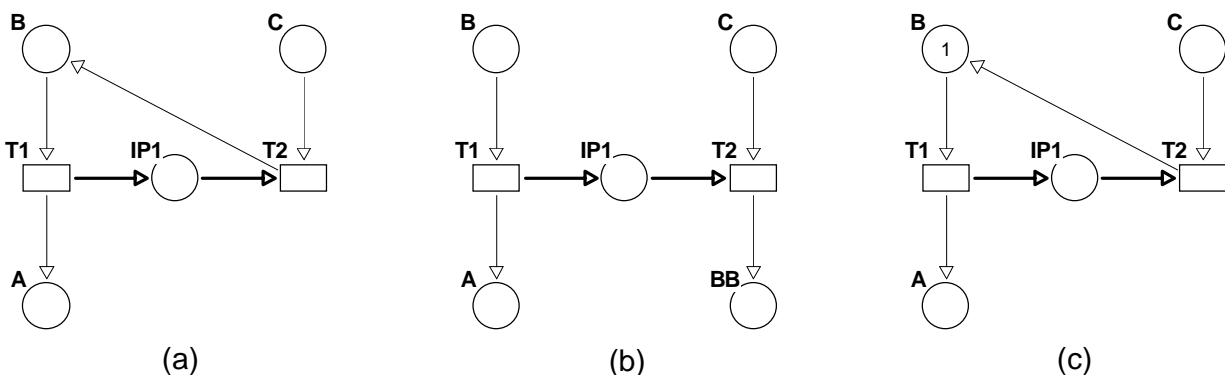
```
T1: A = B
T2: B = C
```


Figure 3: Detection (a) and removal (b) of antidependence

This example shows a simple situation where variable B is used twice, but it can be replaced by two independent variables. The PN model, given in Figure 3(a), contains one P-invariant $IP_1 - B$ with zero tokens. In order to eliminate this P-invariant, evoking a deadlock situation, it is needed either to split IP_1 (that means to change the order of the operations T_1 and T_2 → but this change would bring a result different from the original one) or to split B as shown in Figure 3(b). In this case B becomes an initialized variable (as well as variable C) which matches very well the real situation.

Remark: Please notice that the task is to remove dependencies and corresponding cycles, not only to avoid the deadlock situation (in such case it will be sufficient only to put one token to B as shown in Figure 3(c), that

6

means to initialize the variable B). In the terminology of DDGs we can state that it is possible to remove antidependence and output dependence, but it is not possible to remove flow dependence represented by a normal (not IP) place in the PN model. Figure 3(c) has no more antidependence, but there is a flow dependence due to variable B which disables to fire T_1 and T_2 simultaneously.

> **Example 4:** *consider the following loop program containing an antidependence:*

```
FOR k=1:N
   T1: A(k) = B(k)
   T2: C(k) = A(k+2)
```

The PN model, given in Figure 4(a), contains a P-invariant $A - IP_0$ with minus one token. Please notice that this situation will be represented in DDG by means of a flow dependence with negative value in the dependence vector, but in fact it reflects the same relation as an antidependence.

In order to eliminate the P-invariant $A - IP_0$ it is needed to split variable A as shown in Figure 4(b). A negative number of tokens in the initialized variable A shows a shift of this vector. So the resulting code corresponding to Figure 4(b) is as follows (the fact, that the vector B is initialized too, is omitted):

```
initialize(A(3:N+2))
 FOR k=1:N
   T1: AA(k) = B(k)
   T2: C(k) = A(k+2)
```

> **Rule 2:** *An antidependence is related to a non-live part of a Petri Net. In the case of event graphs it is related to a P-invariant with a non positive number of tokens.*

Remark: There is a polynomial-time algorithm finding a deadlock in event graphs based on Theorem 1 given in [4] (marking is live iff the token count of every directed circuit is positive). If the graph is strongly connected, then it is sufficient to erase all places holding token and then look for existence of directed circuit. If the event graph is not strongly connected, then it is a bit more complicated (one has to compute the strongly connected components), but the problem is still polynomial.
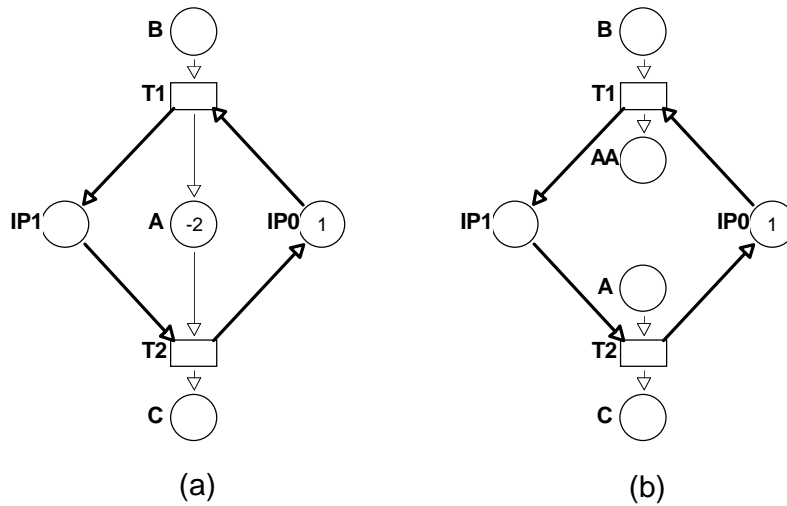
Figure 4: Antidependence in a cyclic algorithm

Example 5: *consider a simple sequence of statements containing an output dependence:*

```
T1:  A = B
T2:  A = C
T3:  D = A
```

The PN model, given in Figure 5(a) demonstrates the place A_x to be implicit to $IP_1 - A_y$. Figure 5(b) shows a resulting model corresponding to the following code:

```
T1:  AA = B
T2:  A = C
T3:  D = A
```

Definition 2 *A place P_x (output place of T_x) is a competitor of P_y (output place of T_y) and P_y is a competitor of P_x iff:*
1) they represent the same variable 2) $T_x \neq T_y$.

Rule 3: *Output dependence is related to the place implicit to its competitor. In the case of event graph: two competitors P_x*
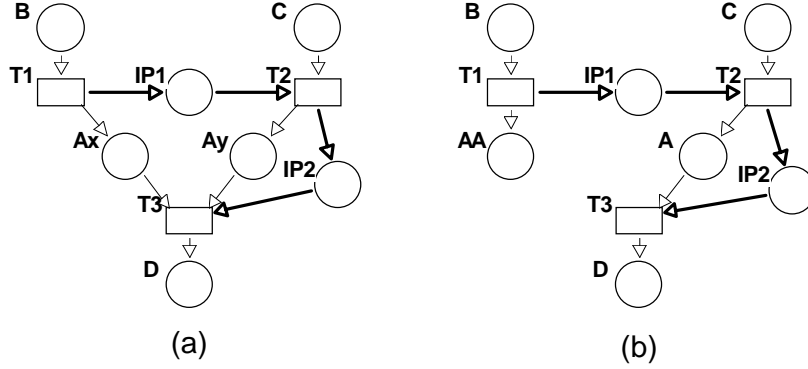
Figure 5: Output dependence

and P_y are related to output dependence iff there is a P-invariant containing both competitors P_x and P_y (one of them with inverted arcs).

The correctness of Rule 3 is proven in the following text.

Example 6: *consider the sequence of statements modelled by a DDG in Figure 6.*

```
T1:  A = B + C
T2:  B = A + E
T3:  A = B
```

This example modelled by the Petri Net in Figure 7(a) contains two antidependencies given by the P-invariant $A_x - B$ and $B - A_y$ and one output dependence $A_x - A_y$. First of all we will try to break antidependecies. The only possibility to disconnect the P-invariant $A_x - B$ is to split the place B and the only possibility to disconnect the P-invariant $B - A_y$ is to split the place A_y. The resulting model is given in Figure 7(b) and reflects the same algorithm as the DDG model given in Figure 6(b). It is important to notice here that there is no more output dependence. The output dependence $A_x - A_y$ disappeared when the P-invariant $B - A_y$ was disconnected. But this is not always the case.
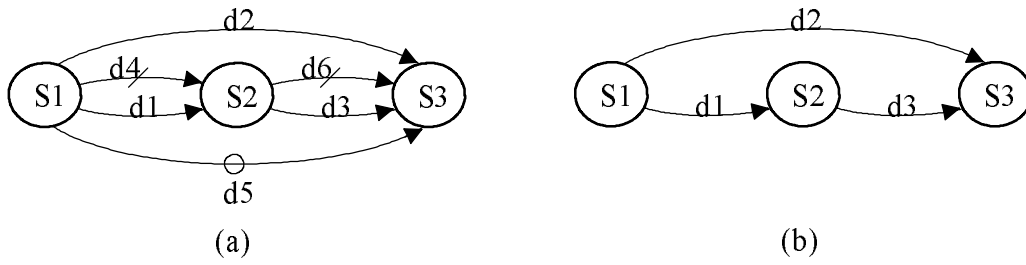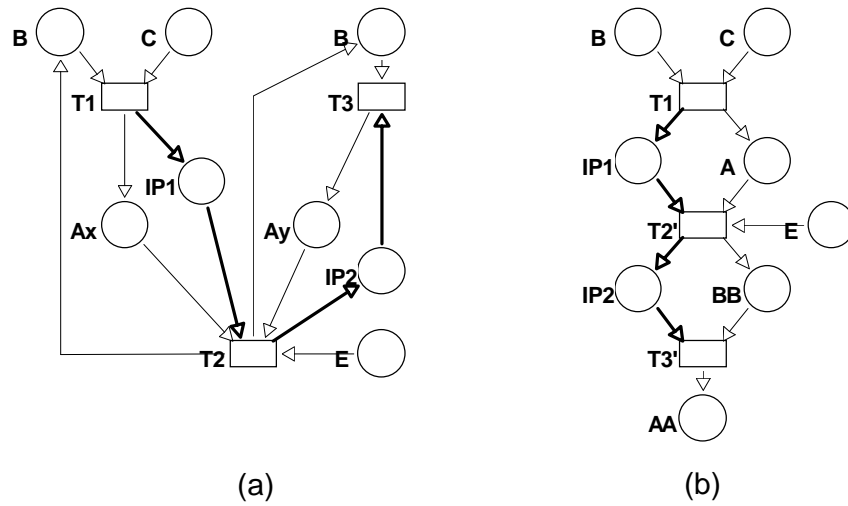
9

Figure 6: Data dependence graph

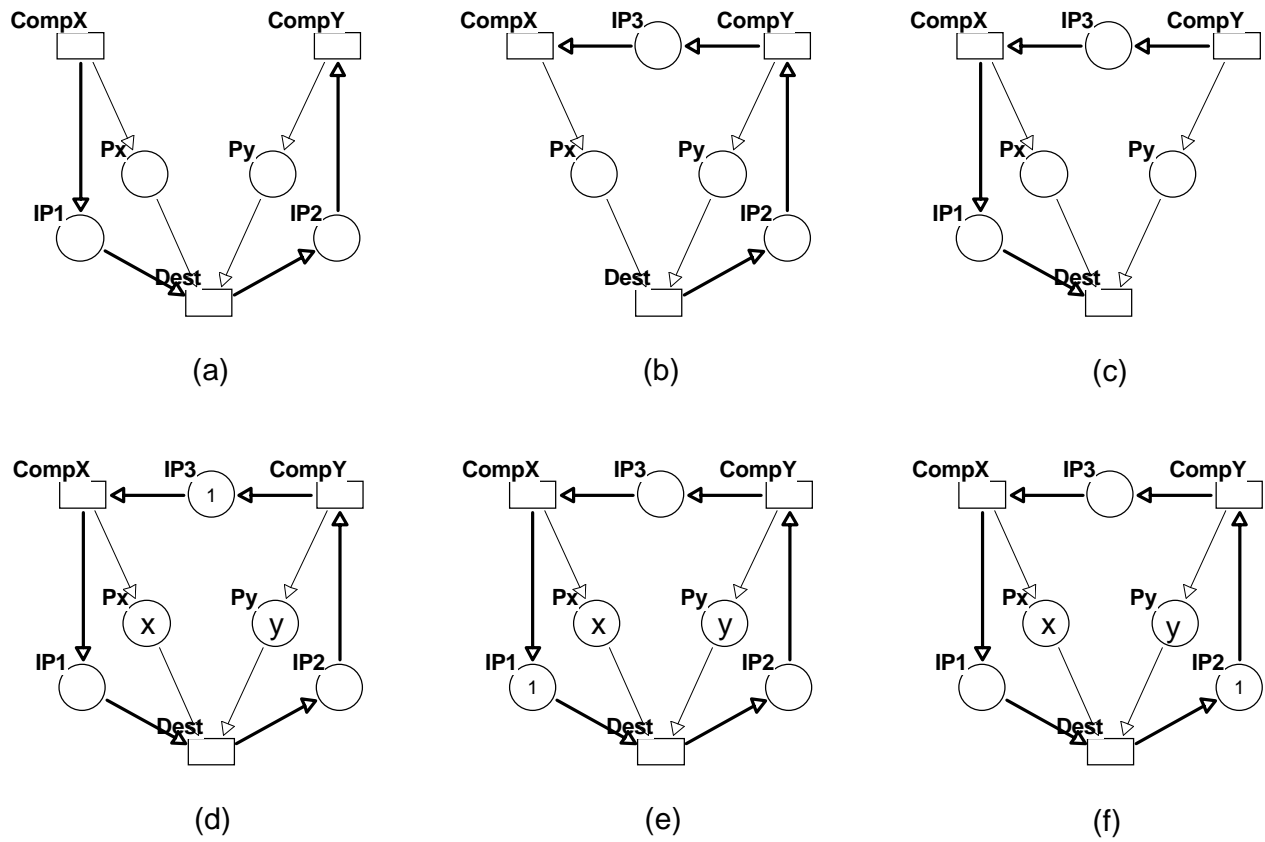Figure 7: Two antidependencies and one output dependence

Figure 8: The six possible instances of an output dependence

Theorem 1 *Let PNO be a Petri net used to model the output dependence in an antidependence free algorithm. Then searching for a P-invariant containing both competitors (P_y with inverted arcs) in PNO is limited to four instances given in Figure 8(c)(d)(e)(f).*

Proof:

- Notice that instances (a)(b)(c) model the three possible combinations of two competitors $CompX, CompY$ and one data destination $Dest$ in acyclic algorithms and instances (d)(e)(f) the three possible combinations in cyclic algorithms.

- Models (a)(b) cannot be taken into consideration because they hold antidependencies. For example Figure 7(a) is one case of instance (a).

- The following holds for instances (c)(d)(e)(f):
  Any additional path (not shown in Figure 8) from $CompX$ to $Dest$ or from $Dest$ to $CompX$ or from $CompY$ to $Dest$ or from $Dest$ to $CompY$ or from $CompX$ to $CompY$ cannot take part in the P-invariant containing both competitors P_x and P_y (P_y with inverted arcs). So these paths need not be present in Figure 8 for the needs of Theorem 1.

- The following holds for instances (c)(e)(f):
  Any additional path from $CompY$ to $CompX$ is implicit to IP_3.

- The following holds for instance (d):
  Any additional path from $CompY$ to $CompX$ with zero tokens creates an antidependence with $IP_1 - IP_2$. Otherwise, with more tokens, it is implicit to IP_3.

□

Theorem 2 *Each of instances (c)(d)(e)(f) has a P-invariant containing both competitors P_x and P_y (P_y with inverted arcs).*

Proof: see Figure 8 □

Theorem 2 implies that we are always able to prove whether place P_y is implicit or not.

Theorem 3 *Let PNO be a Petri net used to model the output dependence in an antidependence free algorithm. Then the following holds for instances (c)(d)(e)(f):*

P_y *is implicit to* $IP_3 - P_x \Leftrightarrow$ *elimination of* P_y *does not change behaviour and results of the modelled algorithm.*

Proof:

- instance c)
  P_y is always implicit $\Rightarrow$ the sequence of IP dependencies shows that the data from the competitor Y are first overwritten by the competitor X and then read by destination

- instance f)
  P_y is implicit $(y \geq x) \Rightarrow$ the same memory location is first assigned by competitor Y and then overwritten by the competitor X
  P_y is not implicit $(y < x) \Rightarrow$ the same memory location is first assigned by competitor X and then overwritten by the competitor Y in one of the successive iterations

- instance e)
  $x = 0$ or $y = 0 \Rightarrow$ there is an antidependence
  $x > 0$ and $y > 0 \Rightarrow$ the transition $Dest$ could be fired and a marking equal to the one in instance c) reached

- instance d)
  $y = 0 \Rightarrow$ there is an antidependence
  $y > 0 \Rightarrow$ the transitions $CompX$ and $Dest$ could be fired and the marking equal to the one in instance c) reached

$\square$

Theorem 4 P_y *is not implicit to* $IP_3 - P_x \Leftrightarrow$ *elimination of* P_x *does not change behaviour and results of the modelled algorithm.*

Proof: From Theorem 3 P_y is not implicit $\Leftrightarrow$ elimination of P_y change the algorithm behaviour $\Leftrightarrow$ data of the competitor Y are used $\Leftrightarrow$ data of the competitor X are not used (see Definition 2).
$\square$

Theorem 5 *An algorithm detecting and removing antidependencies and output dependencies can be schematically written in the following way:*

begin

     1) get PN model from sequential algorithm

     2) eliminate antidependencies

     3) eliminate output dependencies

     4) remove IP-dependencies

     5) save result as DAG (acyclic algorithm)

        or event graph (cyclic algorithm)

end

Proof: see Theorems from this section

□

Example 7: *consider the following loop program with non specified constant z:*

```
FOR k=1:N
    T1: A(k+1) = B(k)
    T2: C(k) = A(k)
    T3: A(k+z) = C(k)
    T4: D(k) = A(k)
```
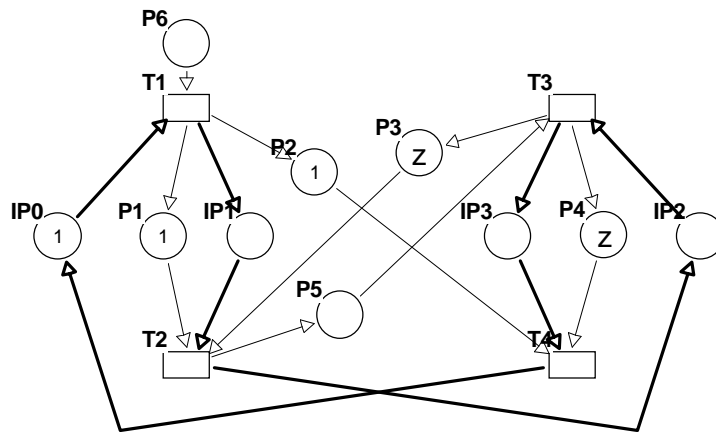

Figure 9: Problem with two competitors and two destinations

14

The PN model, is given in Figure 9. Let us study the output dependencies for various values of the constant z:

- $z \leq 0$
  first of all the antidependence $P_3 - IP_2$ has to be eliminated and then we can look for an output dependence in a new model

- $z = 1$
  P_3 is not implicit to $IP_3 - IP_0 - P1 \Rightarrow P_1$ is eliminated
  P_2 is implicit to $IP_1 - IP_2 - P4 \Rightarrow P_2$ is eliminated

- $z \geq 2$
  P_3 is implicit to $IP_3 - IP_0 - P1 \Rightarrow P_3$ is eliminated
  P_2 is not implicit to $IP_1 - IP_2 - P4 \Rightarrow P_4$ is eliminated

Remark: the case when comparing P_1 to P_3 for destination T_2 corresponds to the instance d) and the case when comparing P_2 to P_4 for destination T_4 corresponds to the instance f).

3   Conclusion

The objective of this article was to build a PN model of problems given either in the form of the equations or in the form of the algorithm.

The basic structural features of algorithms are dictated by their data and control dependencies. These dependencies refer to the precedence relations of computation that need to be satisfied in order to compute the problem correctly. The absence of dependencies indicates the possibility of simultaneous computations. This fact is very important for the automatic program parallelization.

Figure 10 shows a general view of several modeling approaches where information is represented by elipses and methods are represented by rectangles. As shown in Figure 10 both modelling approaches represented by Section 1 (model based on the problem analysis) and Section 2 (model based on the sequential algorithm) lead to the same outcome - representation of data dependencies by event graphs with nonnegative number of tokens in each place. Afterwards for the purpose of parallelism detection the event graph can be reduced as shown in paragraph **??**.

The original features of the article are:
It introduces a new term 'general uniform contraints' leading to negative
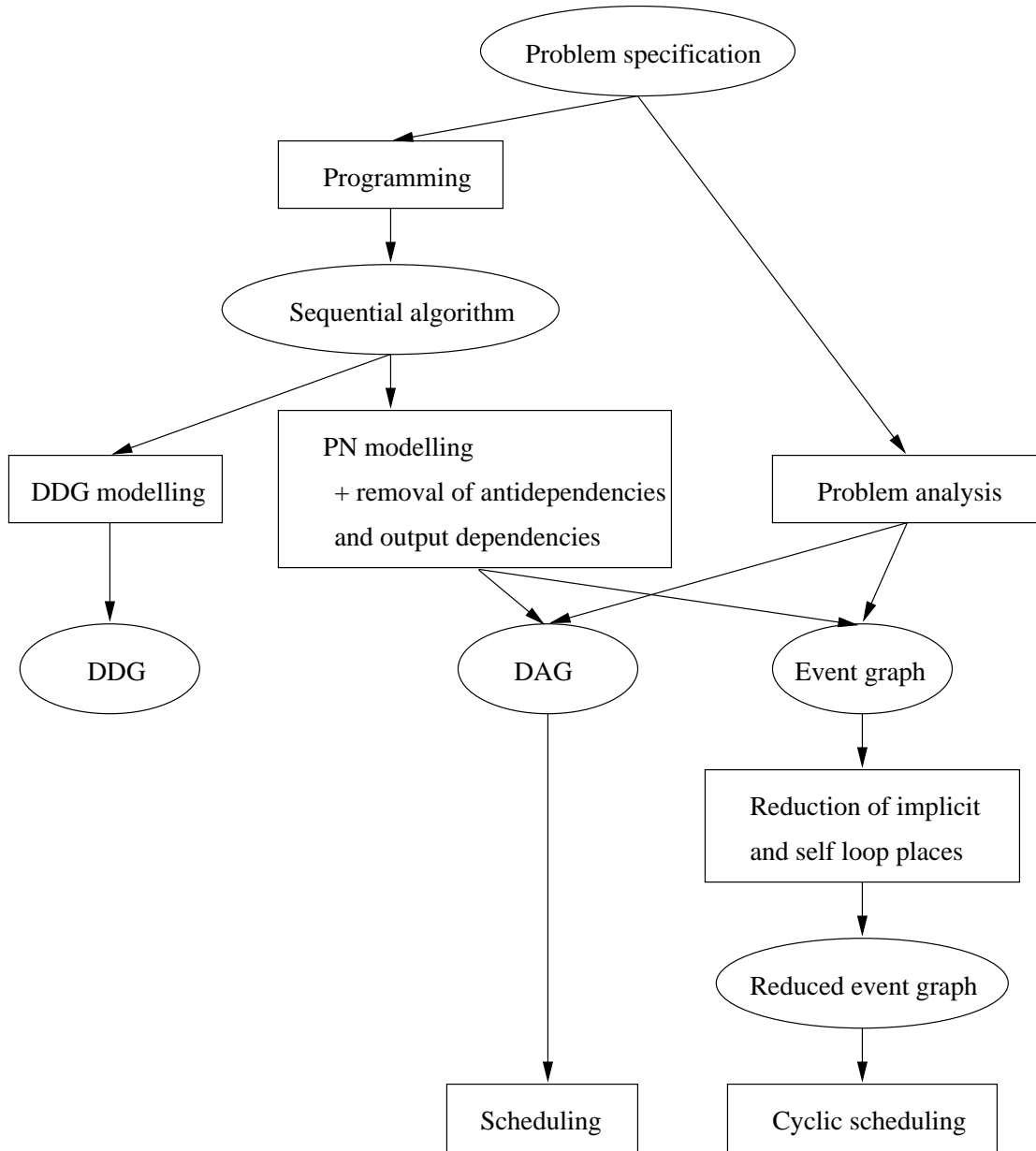
Figure 10: Rough comparison of the two modelling approaches

16

number of tokens and to antidependencies.

It makes an attempt to compare various modelling techniques.

It shows that the data dependencies of iterative problems can be modelled by an event graph.

It shows how to simplify the model by reduction of implicit places and self-loop places, leading to the reduced model which is easier to schedule.

It underlines importance of antidependencies and output dependencies and it shows algorithm tranformations based on PN analysis (removal of dependencies increases a structural parallelism).

As a whole it presents a modeling strategy, which is coherent for noniterative as well as iterative problems.

References

[1] J. Carlier, P. Chrétienne, C. Girault, Modelling Scheduling Problems with Timed Petri Nets, *LNCS 188*, Springer-Verlag (1985).

[2] J. Carlier, P. Chrétienne, Timed Petri Net Schedules, *LNCS 340*, Springer-Verlag (1988) 62-84.

[3] P. Chrétienne, E.G. Cofman, J.K. Lenstra, Z. Liu, *Scheduling Theory and its Applications*, John Wiley & Sons, (1995).

[4] F. Commoner, A.W. Holt, S. Even, A. Pnueli, Marked Directed Graphs, *Journal of Computer and System Science* Vol. 5, (1971) 511-523.

[5] J. Demel, *Graphs (in Czech)*, (SNTL Prague, 1989).

[6] J.A. McHugh, *Algorithmic Graph Theory*, (Prentice Hall, 1990).

[7] D.I. Moldovan, *Parallel Processing - From Applications to Systems*, (Morgan Kaufmann Publishers, 1993)

[8] C. Ramchandani, *Analysis of Asynchronous Systems by Timed Petri Nets*, PhD Thesis, MIT, 1973.

[9] R. Valette, Analysis of Petri Nets by Stepwise Refinement, *J. Comput. Syst. Sci*, Vol. 18 (1979) 35-46.