æ
5
5

Linear Time Algorithm to Find a Minimal Deadlock in a Strongly Connected Free-Choice Net

Peter Kemper

Informatik IV, Universität Dortmund
Postfach 50 05 00, 4600 Dortmund 50, Germany

Abstract. This paper presents an improved algorithm compared to the one given in [7], which finds a minimal deadlock containing a given place p in a strongly connected Free-Choice net (FC-net). Its worst case time complexity is linear in the size of the net. The interest in finding such deadlocks arises from recognising structurally live and bounded FC-nets (LBFC-nets), where finding structural deadlocks efficiently is crucial for the algorithm's time complexity. Employing this new algorithm within [7] LBFC-nets can be recognised in $O(|P|^2|T|)$, which is a reduction by one order of magnitude. Furthermore this marks a lower limit for the complexity of algorithms based on the rank theorem as long as the computation of a matrix rank requires $O(n^3)$.

1 Introduction

Petri Nets have been successfully used for modelling discrete concurrent systems. Within the different classes of Petri Nets the class of FC-nets is theoretically well analysed as far as liveness and boundedness are concerned. In this context the notion of 'deadlock' traditionally refers to a structural not a behavioral deadlock. Since [6] the idea of deadlocks and traps has always been a central point in characterising live FC-nets. Recent research papers reveal a certain progress towards analysis algorithms with polynomial time complexity of low order. In [1] K. Barkaoui and M. Minoux present an algorithm to check liveness for bounded Extended Free-Choice Nets (EFC-nets) and Non Self-Controlling Nets (NSF-nets). This algorithm searches for a strongly connected deadlock which is not a trap and leads to a time complexity of $O(|P|^2|F|^2)$[1].

In [7] an algorithm to recognise structurally life and structurally bounded FC-nets (LBFC-nets) is given. FC-nets are a subclass of EFC-nets, but EFC-nets can be easily transformed into FC-nets, see [3]. This approach has a lower time complexity of $O(|P||T||F|)$. The algorithm is based on the rank theorem given in [4] and decides state-machine-decomposibility of a given FC-net by computing a sufficient set of minimal deadlocks to cover the net. Finding a minimal deadlock for a given place p is crucial in this algorithm and dominates the algorithm's time complexity. The effort for finding a minimal deadlock in

[1] P = set of places, T = set of transitions and F = set of arcs

[7] is $O(|T|(|P| + |T| + |F|))$. The algorithm presented in this paper reduces this effort to $O(|P| + |T| + |F|)$, which is linear in the size of the net. Employing this new algorithm within [7] LBFC-nets can be recognised in $O(|P|^2|T|)$. All algorithms based on the rank theorem require the calculation of the rank of the incidence matrix. As long as the calculation of a matrix rank takes an O(n^3)-algorithm, a further reduction of $O(|P|^2|T|)$ is not possible for these algorithms. In this sense our new algorithm reaches an optimum up to constant factors for recognising LBFC-nets.

This article is structured as follows: Sect. 2 contains basic definitions. In Sect. 3 some previous algorithms and ideas are summarized and by a small example the main idea of the new algorithm is introduced, such that we can present it in Sect. 4 as an improvement of existing algorithms. Section 5 illustrates the behaviour of the new algorithm by an example. Termination is proved in Sect. 6, while Sect. 7 contains a proof of correctness. Section 8 concerns the algorithm's time complexity and finally Sect. 9 finishes the paper with some conclusions and future prospects.

2 Basic Definitions and Theorems

Definition 1 Net.
A net is a triple $N = (P, T, F)$ where

1. P and T are non-empty, finite sets and $|P| = n$, $|T| = m$,
2. $P \cap T = \emptyset$,
3. $F \subseteq (P \times T) \cup (T \times P)$.

The next definition summarizes some well known notions for nets.

Definition 2.
Let $N = (P, T, F)$ be a net.

1. Let $x \in P \cup T$. The preset $\bullet x$ and postset $x\bullet$ are given by
$\bullet x := \{y \in P \cup T | (y, x) \in F\}$, $x\bullet := \{y \in P \cup T | (x, y) \in F\}$.
The preset (postset) of a set of nodes is the union of presets (postsets) of its elements.
2. A path of a net N is a sequence $(x_1, \ldots, x_n)$ of nodes $x_i \in P \cup T, i \in \{1, \ldots, n\}$, such that $(x_i, x_{i+1}) \in F$ for all $i \in \{1, \ldots, n-1\}$.
3. N is a Free-Choice Net (FC-Net) iff $\forall p \in P : |p\bullet| > 1 \Rightarrow \bullet(p\bullet) = \{p\}$.
4. A net $N' = (P', T', F')$ is a subnet of N, $(N' \subseteq N)$, iff $P' \subseteq P, T' \subseteq T$ and $F' = F \cap ((P' \times T') \cup (T' \times P'))$.
5. $P' \subseteq P$ is a deadlock of N iff $P' \neq \emptyset$ and $\bullet P \subseteq P\bullet$. A deadlock is minimal iff it does not contain a deadlock as a proper subset. A deadlock P' is strongly connected iff $N' = (P', \bullet P', F')$ with $F' = F \cap ((P' \times \bullet P') \cup (\bullet P' \times P'))$ is strongly connected.

3 On the Computation of Minimal Deadlocks

This section summarizes ideas and algorithms given in [5] and [7] in order to describe Algorithm 10 in Sect. 4 as a further development. This background allows an easier understanding of the correctness of Algorithm 10.

The precise problem is to find a minimal deadlock in a strongly connected FC-net which contains a given place p. In [5] J. Esparza describes an algorithm which exploits the following characterization of minimal deadlocks.

Theorem 3 [5].
Let $N = (P, T, F)$ be a FC-Net and $D \subseteq P$ a deadlock in N.
D is minimal iff D is strongly connected and $\forall t \in \bullet D : |\bullet t \cap D| = 1$.

The central idea for searching a minimal deadlock is to find a handle for any place of the minimal deadlock (starting with p) that has an input transition not being an output transition.

Definition 4 Handle.
Let $N = (P, T, F)$ be a net with two non-empty sets $S, S' \subseteq P \cup T$, $S \cup S' = P \cup T$ and $S \cap S' = \emptyset$. A path $H = (x_0, x_1, \ldots, x_{n-1}, x_n)$ in N is a handle iff $x_0, x_n \in S$, $x_1, \ldots, x_{n-1} \in S'$, $(x_i, x_{i+1}) \in F$, $\forall i \in \{0, \ldots, n-1\}$ and furthermore $x_i \neq x_j$, $\forall i, j \in \{1, \ldots, n-1\}, i \neq j$.

Informally a handle starts at a node in S, follows a circlefree path in the complement set S' and terminates in S. Note that Def. 4 includes the case $x_0 = x_n$. Given two such sets S and S' a handle always exists in a strongly connected net for any fixed node x_n. The following algorithm computes a minimal deadlock by searching handles for all places that belong to the minimal deadlock.

Algorithm 5. get-minimal-deadlock(N,p), [5] and [7]

Input: N=(P,T,F) strongly connected FC-Net with p $\in$ P.
Output: minimal deadlock D $\subseteq$ P containing p
Initiate: $\hat{P} = \{ p \}$; $\hat{T} = \emptyset$;

Function: get-handle(S,S',N,p,t), see Algorithm 6
This function computes a handle $(x_0, x_1, \ldots, x_{n-2}, t, p)$ with $x_0, p \in$ S
and $x_1, \ldots, x_{n-2}, t \in$ S'. It returns the handle as a set $\{x_0, x_1, \ldots, x_{n-2}, t, p\}$.

```
Program:
begin
    while (∃ p' ∈ P̂ : ∃ t ∈ • p' :  t ∉ T̂ )
    begin
        H := get-handle((P̂ ∪ T̂), (P ∪ T)-(P̂ ∪ T̂),N,p',t) ;
        P̂ := P̂ ∪ ( H ∩ P ) ; T̂ := T̂ ∪ ( H ∩ T ) ;
    end
    D := P̂;
end
```

The set $\hat{\mathrm{P}}$ of places belonging to the minimal deadlock is initiated with $\{p\}$ and successively all places of the computed handles are inserted into $\hat{\mathrm{P}}$. The algorithm terminates if all places in $\hat{\mathrm{P}}$ have handles covering all of their input transitions.

The problem of finding a handle is solved in [7] by Depth-First-Search (DFS). The main idea is that the search stops if an element of S is reached, and that it does a backtrack step if it reaches an element of S' which has been reached before. Due to the fact that no node is checked twice, the search path (stored in a list) does not contain an element of S' twice. The strong connectedness ensures the existence of a handle and the successful termination of DFS. Note that in the following algorithm the search proceeds backwards on the net due to the fact that the last place of the handle is given, while its starting node in S has to be computed. Furthermore the input transition which the handle shall cover is fixed. The number num of a node denotes whether a node was visited before or not.

Algorithm 6. get-handle(S, S', N, p, t),[7]

```
Input: N=(P,T,F) strongly connnected FC-Net, with S,S' ⊆ P ∪ T
    S ∪ S' = P ∪ T, S ∩ S' = ∅, p ∈ S, t ∈ S' and t ∈ • p
Output: Handle H=(x_0, ..., x_{n-2},t,p)
Initiate: i := 1 ; list := empty ; num(p) := 1 ;
    num(x) := 0, ∀ x ∈ S' ;
Function: dfs(v)
begin
    i := i + 1 ; num(v) := i ; append(list,v) ;
    if ( ∃ w ∈ •v : w ∈ S )                  { element of S is reached }
    then append(list,w) ; return Yes ;        { search stopped }
    forall ( w ∈ •v with num(w) = 0 )         { for unvisited elements of S'}
    begin                                     { recursive search }
        if ( dfs(w) = Yes )
        then return Yes ;
    end
    delete(list,v) ;
    return No ;
end

Program:
begin
    append(list,p) ;
    dfs(t) ;
    Output list ;
end
```

Algorithm 5 employing Algorithm 6 leads to a time complexity of $O(|T|(|P| + |T| + |F|))$, because at most $|T|$ handles have to be computed for a deadlock and computing a handle requires $O(|P| + |T| + |F|)$; see [7] for details. Is there a possibility to reduce this time complexity? In Algorithm 6 those paths are regarded as unsuccessful which lead back to a node w in S', which was visited before (num(w) > 0). These "unsuccessful" paths can be suitable as handles later on. The following example shows the effect we are interested in.

Example 1. Figure 1 shows a strongly connected FC-net. Let p be the place a minimal deadlock shall be computed for. The numbering of nodes show the num-values given by Algorithm 6 which started on p with transition a. If we focus on the list content whenever a node is reached that was visited before or when p is reached, we get the following four list contents:

1. p a b c d e f
2. p a b c g h
3. p a b c g i j k
4. p a b c g l p

List content 4 gives the handle we are looking for. In Algorithm 5 three further handles have to be computed. Two starting at place g, one for input transition h the other one for input transition i. The third one starts in p for transition h and becomes irrelevant after covering h with a handle from g. The interesting point is, that these handles were computed within the first get-handle call but have been regarded as not interesting. Part of list content 3 hold the corresponding information for the required handle (g i j k b). But the information for the handle (g h f a) is spread over list contents 1 and 2. Furthermore the list content 1 gives an example of the irrelevant path (c d e f a) which starts at a transition.

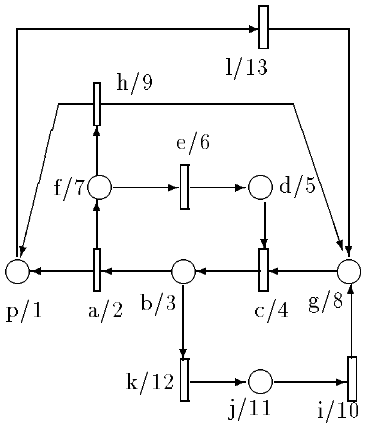

Fig. 1. Strongly connected FC-net after applying Algorithm 5 for place p, node description: <id/num-value>

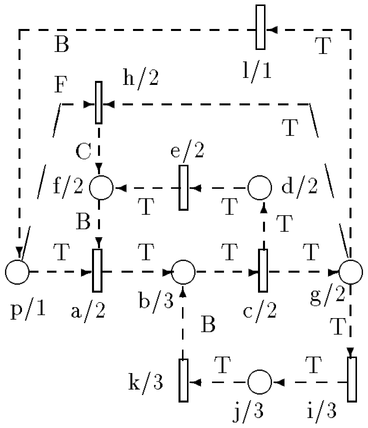

Fig. 2. Corresponding DFS-arcs after applying Algorithm 10 for place p, node description: <id/ll-value>

This small example shows that some of the knowledge we earn from former unsuccessful attempts is useful later on. A more sophisticated way of computing a minimal deadlock should exploit this knowledge. Previously unsuccessful paths must be labelled in such a way, that handles contained in these paths, are tractable without further searching. An algorithm which follows this idea naturally splits up into 3 phases:

1. Searching for a handle,
2. marking a handle as part of the deadlock and
3. evaluating old knowledge in order to mark directly those handles which consist of paths computed before.

This approach leads to an algorithm that computes a minimal deadlock within a single DFS-run. A detailed description of this algorithm follows within the next section.

4 Improved Algorithm to Find a Minimal Deadlock

This section presents an algorithm to compute a minimal deadlock for a given place of a strongly connected FC-net within a single DFS-run. Since the search for handles proceeds backwards on the net, we define DFS-arcs in order to avoid confusion on the direction of arcs.

Definition 7 DFS-arc. Let $N = (P, T, F)$ be a net.
f = (v,w) is a DFS-arc if (w,v) $\in$ F.

By analogy with DFS-arc we use terms like DFS-successor, DFS-path etc. with prefix DFS to stress that we use opposite directions as far as arcs of the net are concerned.

As imposed in Sect. 3 a labelling of nodes (resp. a classification of DFS-arcs) is necessary to utilize knowledge about already computed DFS-paths. According to the numbering of nodes in Algorithm 6 we separate four types of DFS-arcs. Regard a dfs(v)-call[2] which checks a DFS-arc (v,w).

1. T(ree)-arcs
   This type of DFS-arc leads to a node which has not been reached before.
   Formal criterion : num(w) = 0.
2. F(orward)-arcs
   This type of DFS-arc leads to a node which was already visited and which is reachable via a DFS-path of T-arcs starting at v. This means w was reached by performing dfs(w') for a T-arc(v,w') beforehand.
   Formal criterion : num(w) > num(v).
3. B(ack)-arcs
   This type of DFS-arc leads back to a node w from which v was reached by T-arcs and whose dfs(w)-call has not terminated yet. It closes a circle.
   Formal criterion: num(w) < num(v) $\wedge$ w $\in$ list.

[2] Function dfs corresponds to function get-handle in the context of Algorithm 10.

4. C(ross)-arcs
   This type of DFS-arc leads to a node w whose dfs(w)-call terminated before v was found. This means that no DFS-path of T-arcs from w to v exists.
   Formal criterion: num(w) $<$ num(v) $\wedge$ w $\notin$ list.

This classification of DFS-arcs is taken from [8], [10], where it is used for finding strongly connected components. Figure 2 shows the classification of DFS-arcs according to the net in Fig. 1, the ll-values in the node description refer to Def. 9. T-arcs form a tree on the net, while a B-arc closes a circle within this tree and C-arcs cross over subtrees, see also Fig. 5. An F-arc is a shortcut for a DFS-path of T-arcs. In FC-nets only one type of F-arcs can occur.

Lemma 8. *Apply Algorithm 6 on a strongly connected FC-net.*
If (v,w) is an F-arc $\Rightarrow$v is a place and w is a transition.

Note that an F-arc is a DFS-arc and its corresponding net arc has the opposite direction.

Proof. Assume $v \in T, w \in P$
$\Rightarrow \exists x \in \bullet v : x \neq w$ and $\exists y \in w\bullet : y \neq v$ with (v, x) and (y, w) are T-arcs (reachability by T-arcs)
$\Rightarrow |w \bullet| > 1$ and $\{w, x\} \subseteq \bullet(w\bullet)$ contradicting FC-condition. □

The information which shall be stored at a node is its low-link value. This value for a node x holds the minimal num-value of any node reachable from x via a DFS-path on the net that consists of an arbitrary number of T- and/or C-arcs and finally one B-arc. Note that this definition differs from a similar definition in [10] with respect to C-arcs.

Definition 9 low-link value (ll-value).

$$
\begin{aligned}
ll(x) := min(\{ \ num(x) \ \} &\cup \{ \ num(z) | (x, z) \text{ is B-arc } \} \\
&\cup \{ \ ll(y) | (x, y) \text{ is T-arc or C-arc } \})
\end{aligned}
$$

F-arcs cannot reduce the low-link value of a node any further, because any node reachable by an F-arc is also reachable via a DFS-path of T-arcs. This ensures that any num-value reachable by an F-arc is already considered.

Figure 2 corresponds to the net in Fig. 1 and shows DFS-arcs according to a search starting at place p for transition a. The labelling of a node gives its identifier and its ll-value. The corresponding num-values are the same ones as in Fig. 1. It displays the situation when handle (p a b c g l p) is found. Missing handles (g h f a) and (g i j k b) reveal constant ll-values 2, resp. 3. So marking handles following a DFS-path with constant ll-values is promising. This is not sufficient in general, because this DFS-path stops at a B-arc (v,w), with w not necessarily being a suitable termination node. Furthermore this DFS-path might get trapped in a circle. An appropriate strategy turns out to be "follow a DFS-successor with minimal ll-value" which the following algorithm demonstrates.

Algorithm 10.

```
Input: N=(P,T,F) is a strongly connected FC-Net with p ∈ P
Output: Minimal Deadlock D with p ∈ D
Initiate:
    i := 0 ; max-num := 1 ; list := empty ; D(p) := True ;
    ∀ x ∈ T ∪ P \{p} : num(x) := 0 ; ll(x) := 0 ; D(x) := False ;

Program:
begin
    get-handle(p) ;
    ∀ x ∈ P, D(x) = True : output x ;
end

Function: get-handle(v)
begin
    i := i + 1 ; num(v) := i ; ll(v) := i ; append( list, v) ;
    forall w ∈ • v
    begin
        if num(w) = 0                                   {T-arc}
        then get-handle(w) ;
            ll(v) := min(ll(v),ll(w)) ;
            if D(w) = True ∧ v ∈ T              { handle was found }
            then delete(list,v) ; return ;      { stop search for transition }

        if num(w) > num(v)                              {F-arc}
        then if num(v) ≤ max-num
            then eval-old(w) ;                  { to complete handle }

        if 0 < num(w) < num(v) ∧ w ∈ list               {B-arc}
        then ll(v) := min(ll(v), num(w)) ;
            if D(w) = True                      { handle found }
            then mark-handle() ;
                if v ∈ T                        { stop search for transition }
                then delete(list, v) ; return ;

        if 0 < num(w) < num(v) ∧ w ∉ list               {C-arc}
        then ll(v) := min(ll(v),ll(w)) ;
            if num(w) ≤ max-num
            then eval-old(w) ;                  { to complete handle }
                mark-handle() ;
                if v ∈ T                        { stop search for transition }
                then delete(list, v) ; return ;
    end forall
    delete(list, v) ;
end function
```

```
Function: eval-old(v)
begin
    if ( D(v) = True or v ∈ list)
    then return ;                                  { stop recursion }
    D(v) := True ;
    if v ∈ T
    then choose one w ∈ • v with minimal ll(w) ;   { need not be unique }
        eval-old(w) ;
    else ∀ w ∈ • v : eval-old(w) ;
end function

Function: mark-handle()
begin
    forall ( x ∈ list with max-num < num(x) )
    begin
        D(x) := True ;                             { mark element }
        if x ∈ P                                   { exploit knowledge }
        then ∀ y ∈ • x, num(y) > 0, y ∉ list : eval-old(y) ;
    end forall
    max-num := i ;                                 { update max-num }
end function
```

According to the three phases mentioned in Sect. 3 the algorithm splits into three functions:

get-handle to search for a handle with DFS and classify DFS-arcs
It checks fresh, yet unexplored nodes and converts them to old, explored ones. If it reaches a node, which terminates a handle (by a B-arc) or which can surely be completed to a handle (by C- or F-arc), function get-handle makes sure that this handle gets marked by function mark-handle. If necessary it completes the handle with function eval-old. Furthermore it tries to cover all DFS-successors of places, i.e. their input transitions, with handles (according to Def. of deadlocks). If it does not succeed, it leaves this to function eval-old. At a transition it tries to ensure that only one DFS-successor, i.e. only one input place, becomes element of the deadlock (according to Theorem 3).

mark-handle to mark elements of a handle
Along the new handle it invokes function eval-old for those DFS-successors of places, for which function get-handle did not succeed beforehand. This ensures that already explored input transitions of these places are covered with handles by function eval-old.

eval-old to mark handles tractable from previously unsuccessful attempts
Its duty is to follow handles on already explored parts of the net and to mark them. Under the assumption that its current node v belongs to a handle (which turns out to be true), it follows all DFS-successors for $v \in P$

(according to Def. of deadlocks) or only one DFS-successor for v $\in$ T (according to Theorem 3). It chooses the one DFS-successor in such a way that it cannot get trapped in a circle.

Attributes of a node v are:

num (integer) holds num-value,
ll (integer) stores ll-value and
D (boolean) denotes membership of minimal Deadlock D, resp. $\bullet$ D.

Whenever a node x is called marked, this means: x is marked D(x) = True.
The variables used in Algorithm 10 have the following meaning:

i (integer) is a counter to give any reached node a unique number.
list stores nodes of the recursion stack of function get-handle.
max-num (integer) holds the maximal number of nodes reached when the last handle was found. It has two duties. On the one hand it splits list in such a way that all nodes in the first part (num(x) $\leq$ max-num) are marked with D(x)=True and all nodes in the second part (num(x) $>$ max-num) remain D(x)=False until a handle is found. On the other hand it splits those nodes exploitable as old knowledge (num(x) $\leq$ max-num) from nodes found too recently to be exploited (num(x) $>$ max-num). Note that node x with num(x)=max-num is not necessarily element of list.

5 Example Computation

This section illustrates the behaviour of Algorithm 10 by an example. The following lemma simplifies the naming of nodes.

Lemma 11. *Any node v can at most cause one get-handle(v)-call and obtains a unique number num(v) within this call.*

Proof. In both cases, the get-handle(p)-call in program and the get-handle(w)-call in get-handle(v), num(p)= 0, resp. num(w)=0 holds beforehand. But within any get-handle(v), num(v) is immediately set to the incremented counter i. This has the effect, that num(v) is unique. It is not set anywhere else and i is used exclusively for this purpose. A recursive call get-handle(v) with num(v) $>$ 0 is not possible. □

Example 2. Figure 3 shows a strongly connected FC-net. Algorithm 10 computes a minimal deadlock for place 1. Nodes reached by the algorithm are labelled by their num-values, which are unique according to Lemma 11. Unlabelled nodes are not reached and have a num-value 0 which is omitted in Fig. 3. Figure 4 gives the corrsponding DFS-arcs. B-arcs and C-arcs are labelled B, resp. C. F-arcs do not occur. Unlabelled arcs denote T-arcs. Nodes in Fig. 4 are labelled <num-value/ll-value>. Note that these ll-values are the final ones. During computation temporarily higher ll-values occur, which are represented in the list contents

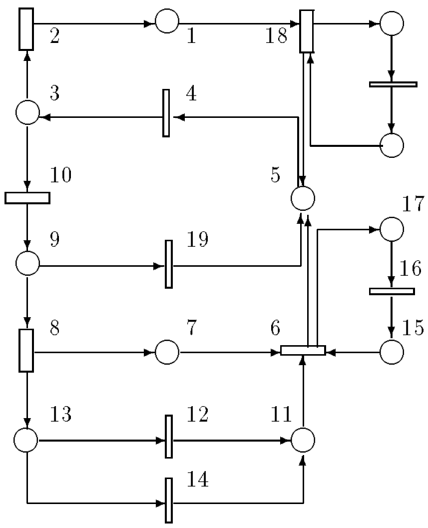

Fig. 3. Strongly connected FC-net

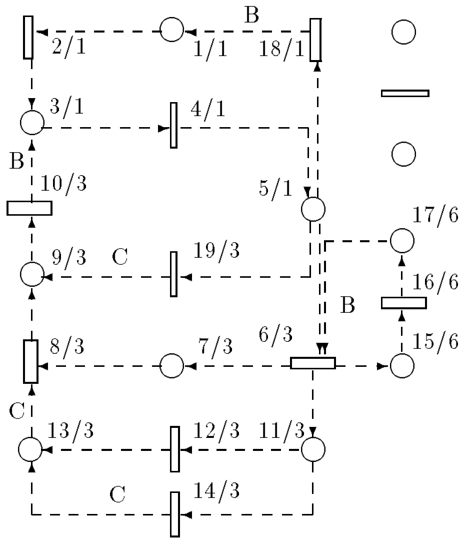

Fig. 4. Corresponding DFS-arcs after computation of a minimal deadlock containing place 1

below. Note that list keeps track of the get-handle-recursion. Let us highlight certain steps of the computation by giving the list content with <num-value/ll-value> and additional explanations:

start max-num = 1, list empty, marked nodes { 1 }

B-arc(10,3) list content: (1/1,2/2/,3/3,4/4,5/5,6/6,7/7,8/8,9/9,10/10)
The num-value 3 is catched from node 3 and by termination within recursion of get-handle it is inherited as ll-value to nodes 10, 9, 8, 7 and 6. get-handle steps back to transition 6.

C-arc(13,8) list content: (1/1,2/2,3/3,4/4,5/5,6/3,11/11,12/12,13/13)
The ll-value 3 from node 8 is inherited to nodes 13,12 and 11. get-handle steps back to place 11.

C-arc(14,13) list content: (1/1,2/2,3/3,4/4,5/5,6/3,11/3,14/14)
The ll-value 3 from node 13 is inherited to node 14. get-handle steps back to transition 6.

B-arc(17,6) list content: (1/1,2/2,3/3,4/4,5/5,6/3,15/15,16/16,17/17)
The num-value 6 from node 6 is inherited as ll-value to nodes 17,16 and 15. get-handle steps back to place 5.

B-arc(18,1) list content: (1/1,2/2,3/3,4/4,5/5,18/18)
The first handle is found. It must be found via a B-arc, because 1 is the only node that is marked and max-num = 1. The num-value 1 from node 1 is inherited as ll-value to nodes 18,5,4,3 and 2. The mark-handle-call marks nodes 2,3,4,5 and 18. max-num is set to 18. eval-old(6) is called, which either marks { 6,7,8,9,10 } or { 6,11,12,13,14,8,9,10 }. Note that at transition 6 the recur-

sion sees ll-values ll(7)=ll(11)=3 and ll(15)=6, thus it follows either place 7 or place 11. Assume that place 11 is chosen. Obviously choosing a DFS-successor with non-minimal ll-value, i.e. place 15, would lead eval-old into a circle, which corrupts correctness. Note that the get-handle recursion returns at transition 18 without checking further input places due to D(1)=True. At place 5 get-handle starts searching for a handle covering input transition 19.

C-arc(19,9) list content: (1/1,2/2,3/3,4/4,5/1,19/19)
$9 \leq$ max-num, thus a handle is found. eval-old(9) terminates right away due to D(9)=True. mark-handle marks node 19. max-num is set to 19. get-handle steps back to place 1 and terminates. The ll-value of any node corresponds to the ll-values given in Fig. 4.

Finally the set of marked nodes is { 1,2,3,4,5,6,8,9,10,11,12,13,14,18 } which results in the minimal deadlock { 1,3,5,9,11,13 }. At transition 18 it is not neccessary to regard the unlabelled places, because only one input place of a transition must be covered by a handle. This is satisfied by place 1, thus any further input place of transition 18 need not be checked which saves some work.

6 Termination of Algorithm

Prooving termination makes use of the two following observations:

Lemma 12. list *is an ordered list by increasing numbers* num*.*

Proof. A node v is appended to list only at the beginning of its get-handle(v) call. As num-values are unique by Lemma 11 and increasing, list is automatically sorted. □

Lemma 13. *Any node v can cause recursion in eval-old(v) at most once.*

Proof. D(v)=False is a precondition for recursion in eval-old(v). D(v):=True is set ahead of recursion. Reset D(v) does not happen, because v is set D(v):=False only by initialisation. □

Theorem 14. *Algorithm 10 terminates for any finite net N.*

Proof. 1. The recursion in eval-old terminates due to Lemma 13 and to the finiteness of the net.
2. mark-handle terminates because list is finite (ensured by the finiteness of the net, Lemma 11 and Lemma 12) and eval-old terminates.
3. get-handle terminates due to termination of eval-old, mark-handle and finite depth of recursion of get-handle ensured by Lemma 11.
4. program terminates due to termination of get-handle(p)-call.

□

7 Correctness of Algorithm

There are at least two ways to prove correctness of Algorithm 10. One way is to show that its output is a deadlock and furthermore a minimal deadlock. Another way is to show that Algorithm 10 realizes the idea of Algorithm 5, which exhaustively calculates handles to cover all input transitions of every place that belongs to the deadlock. In this section we follow the second way, i.e. we show first that eval-old marks a set of handles and as a second, main step that the "handles" found in get-handle actually are handles and no input transition of a deadlock place remains uncovered.

The following properties of Algorithm 10 turn out to be useful for proving its correctness. As mentioned in Sect. 4 max-num is used to separate list in a first part with marked nodes that already belong to a handle and a second part with nodes that might belong to a handle. These nodes are unmarked when appended to list and marked as soon as a handle is found.

Lemma 15. $x \in$ list $\wedge$ *num(x)* $\leq$ max-num $\Rightarrow$*D(x)* = *True*

Proof. by induction on updates of max-num (max-num := i)
Start: by initialisation max-num = 1 and list = empty, which fulfills the lemma. append(list,p) in get-handle(p) does not disturb the lemma due to D(p)=True by initialisation. As num(p) = 1, Lemma 11 and Lemma 12 ensure that for any further append(list, x) within get-handle(x), num(x) > 1 holds. Thus the lemma holds until the first update of max-num.
Step: update of max-num occurs in mark-handle
if x $\in$ list $\wedge$ num(x) $\leq$ max-num $\Rightarrow$D(x) = True by assumption
if x $\in$ list $\wedge$ num(x) > max-num $\Rightarrow$D(x):= True in mark-handle before the update.
Thus $\forall$ x $\in$ list : D(x) = True at update of max-num. □

Lets regard the effect of Algorithm 10 from a node's point of view. Let v be a node different from p. At initialisation num(v)=0 and v belongs to an unexplored part of the net. Once reached by get-handle, it gets num(v) > 0, ll(v) = num(v), v $\in$ list and num(v) > max-num. If a handle is found while v $\in$ list, i.e. get-handle(v) has not terminated, then v gets marked in mark-handle and num(v) $\leq$ max-num holds after update of max-num. Otherwise, if no handle is found while v $\in$ list, get-handle(v) terminates with D(v) = false. Once v is deleted from list, v $\notin$ list holds forever and v belongs to an old, explored part of the net. If v is still unmarked, it can be reached and marked once by eval-old, which just operates on explored parts of the net. During computation, the net separates in unexplored and explored parts. This is formalized in the following notation.

Definition 16 $S_{list}, S_{old}, S_{new}$.
$S_{list} := \{x | x \in \text{list} \vee D(x) = True)\}$, $S_{old} := \{x | x \notin S_{list} \wedge num(x) > 0\}$, and $S_{new} := \{x | x \notin S_{list} \wedge num(x) = 0\}$.

Obviously $S_{list} \,\dot\cup\, S_{old} \,\dot\cup\, S_{new} = P \cup T$ holds. The following lemma states that S_{list} forms a borderline between explored parts (S_{old}) and unexplored parts (S_{new}).

Lemma 17. *In computation of Algorithm 10 no DFS-arc (v,w) exists with $v \in S_{old}$ and $w \in S_{new}$.*

Proof. Assume the contrary: DFS-arc (v,w) with v $\in S_{old}$ and w $\in S_{new}$ exists. Due to num(v) > 0 and v $\notin$ list, get-handle(v) has occured and terminated to reach the assumed situtation.

Case v $\in$ P There is no way to hinder get-handle(x) for all x $\in \bullet$ v during computation of get-handle(v).
$\Rightarrow$num(w) > 0, w $\notin S_{new}$, contradicting the assumption.

Case v $\in$ T num(v) > 0, num(w) $= 0$ by assumption, i.e. no get-handle(w) occured. This could happen only in case of D(w') = True for some other w' $\in \bullet$ v. This is caused by finding a suitable B- or C-arc in recursion from get-handle(w') and calling mark-handle. During get-handle(w'), v $\in$ list and w' $\in$ list hold. Thus w' is marked by mark-handle. Hence, if num(v) $>$ max-num, then v gets marked in mark-handle; otherwise Lemma 15 ensures that D(v) = True holds already. So v $\notin S_{old}$, contradicting the assumption.

□

Obviously get-handle operates on S_{list} and S_{new}, mark-handle on S_{list} and eval-old on S_{old}.

Strong connectedness has an impact on the algorithm's correctness. Figure 5 shows a typical situation within a computation. list contains nodes (a,b,c,d,e) with D(a)=D(b)=True and D(c)=D(d)=D(e)=False. Dotted lines to the unexplored part denote the possibility of further arcs, which were not followed yet. Triangles $\bar{b}, \bar{c}, \bar{d}, \bar{e}$ below nodes denote subtrees of nodes which were reached by T-arcs and whose get-handle-calls already terminated. Strong connectedness requires that these subtrees have B- or C-arcs as "exits". Note that C-arcs can only lead to subtrees previously explored. The B-arc from subtree $\bar{b}$ completes a handle, because it leads back to the starting node a. num(b) $<$ max-num $<$ num(c) holds, since get-handle(c) occurs after the update of max-num. The B-arc in subtree $\bar{c}$ does not complete a handle because it leads to c which is on list, but not yet element of a handle (D(c)=False). The C-arc (d',c') with d' $\in \bar{d}$ and c' $\in \bar{c}$ does not reach a subtree, which can lead to an element of a handle. This is recognised by max-num $<$ num(c'). The F-arc does not reach an element, from which a handle can yet be found, because the B-arc of $\bar{e}$ does only lead to e which is not element of a handle (D(e)= False). If this B-arc would reach further back, say back to b, a handle would run through c,d,e and D(e)=True. By D(e)=False it is recognizable that from the DFS-successors of e so far no handle can be reached.

Lemma 18. *Regard a situation in Algorithm 10 where eval-old is called in get-handle or mark-handle. Let S := S_{list}, S' := $S_{old} \cup S_{new}$ and $v \in (\bullet d) \cap S_{old}$ for $d \in$ list. Then eval-old(v) marks a handle H covering v and handles for all places marked by eval-old(v).*

Proof. Note that S is defined ahead of the eval-old(v)-call, such that this set does not automatically get new elements when nodes get marked within eval-old(v).

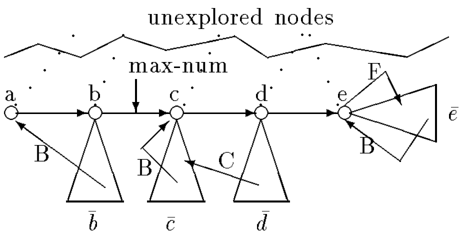

Fig. 5. Typical situation occuring in computation

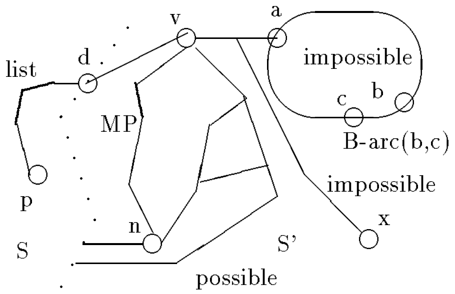

Fig. 6. Possible and impossible behaviour of eval-old(v)-call

Assume that in eval-old the multiple recursion at places is ordered by ll-values of their DFS-successors in a way, that eval-old(w) for a DFS-successor with minimal ll-value is called first. This causes eval-old to follow first a DFS-path on the net directed by its minimal ll-values. This DFS-path is called minimal path MP. First we show that MP is a handle.

- No circles in MP
  The recursion starting at eval-old(v) cannot proceed over nodes of S. Thus Lemma 17 and v $\in S_{old}$ ensure that MP cannot contain a node x $\in S_{new}$. Furthermore x $\notin$ list holds for all nodes x in MP. Thus their get-handle(x)-call has terminated and their ll-values are not changed anymore.
  Obviously function get-handle reflects Def. 9, thus for any node x in MP, num(p) = ll(p) = 1 $\leq$ ll(x) $\leq$ num(x) holds.
  Regard a DFS-arc (y,z) within MP. The case z $\in$ S is uncritical, because it finishes MP in S. Let z $\in$ S' then. Def. 9 states ll(y)=ll(z) for a T- or C-arc and ll(y)=num(z) for a B-arc. For an F-arc (y,z) z is reachable via a DFS-path of T-arcs. Thus a T-arc (y,z') with ll(z') $\leq$ ll(z) exists. By (y,z) in MP and Def. 9 we get ll(y)=ll(z)=ll(z'). For a node z apart from node p (with num(p)=1) strong connectedness requires ll(z) $<$ num(z) for final ll-values, because otherwise DFS-predecessors of z would not be reachable from z. Thus at a B-arc we get ll(y) $>$ ll(z). B-arcs are the only arcs that close circles within the tree of T-arcs. Thus if MP contains a circle, this circle contains at least one B-arc. Constant ll-values at T-, C- and F-arcs in combination with decreasing ll-values at B-arcs contradict the existence of a circle-closing arc in MP, e.g. in Fig. 6 circle(a,b,c) requires ll(a) $\geq$ ll(b) $>$ ll(c) $\geq$ ll(a).
- Termination in S
  Assume the contrary: an eval-old-call terminates in some x $\in$ S', cf. Fig. 6. Due to strong connectedness and Lemma 17, x must have at least one DFS-successor y with num(y) $>$ 0. As x $\notin$ S, eval-old(x) would proceed in MP with the DFS-successor of minimal ll-value contradicting the assumption.

Altogether MP does not contain a circle and terminates in S. Hence, if d $\in$ list $\subseteq S$ is added to MP, handle H is MP regarded in opposite direction.

If S is enlarged by MP, then calling eval-old for non-minimal ll-value DFS-successors at places equals the starting situation and the same argumentation can be applied again. Hence, under the assumption of ordering the multiple recursion by ll-values, a correct marking of handle H and handles for all places marked by eval-old is ensured.
In eval-old recursive eval-old-calls at places can occur in an arbitrary order. Let $call_{MP}$ denote the call following MP and $call_{arb}$ be an arbitrary call which is caused by the multiple recursion within $call_{MP}$ and which follows arbitrary, maybe non-minimal ll-values. It is possible that $call_{arb}$ reaches a node n ahead of $call_{MP}$, cf. Fig. 6. This does not corrupt correctness, because $call_{arb}$ invokes eval-old(n) which is identical to the effect of $call_{MP}$ reaching n first. In fact $call_{arb}$ picks up the role of $call_{MP}$ and proceeds at node n, while $call_{MP}$ terminates when reaching n in second place. Thus finally the same set of nodes get marked. Therefore the order is not required for correctness and can be neglected. The multiple recursion at places reached by eval-old ensures, that all input transitions of these places are covered by handles and get marked. □

Theorem 19. *Algorithm 10 is correct, i.e. its output is a minimal deadlock.*

Proof. For correctness we have to show that

1. whatever node x is marked D(x) := True within mark-handle or eval-old, it actually belongs to a handle and
2. any input transition of a place d with D(d)=True is covered by a handle.

If these requirements are fulfilled, Algorithm 10 behaves just like Algorithm 5, which exhaustively calculates handles to cover all input transitions of any place which belongs to the deadlock.
at 1) Induction on finding handles in get-handle. Handles are found in 3 situations:

1. B-arc (v,w) with D(w)=True.
2. C-arc (v,w) with num(w) $\leq$ max-num.
3. F-arc (v,w) with num(v) $\leq$ max-num.

Induction assumption:
Any node, that gets marked in mark-handle or eval-old until the i-th handle is found in get-handle, belongs to a handle.
Induction start:
The first handle is found by a B-arc (v,p) due to the fact that p $\in$ list, D(p) = True and max-num = num(p) = 1. In this case list obviously contains the new handle. So marking all elements of list except p (max-num=1) is correct. According to Lemma 18 handles are marked by calling eval-old(y) for all nodes y $\in \bullet x$ forall nodes x $\in$ list with num(x) > max-num. These handles terminate in S_{list} which is correct because in this case $S_{list} = \{$ x | D(x)=True $\}$. Thus any node that gets marked belongs to a handle and this handle gets marked completely.
Induction step:

1. Handle found with B-arc (v,w)
   w $\in$ list and marked correctly (by induction assumption), i.e. w is element of a handle. Lemma 15 ensures that elements of list with a num-value $\leq$ max-num are marked. Thus all elements of list with num $>$ max-num form the new handle and marking these nodes is correct. As far as eval-old-calls within mark-handle are concerned, the same argumentation as for the induction start holds again. Thus all nodes marked D(x)=True belong to a handle.
2. Handle found with C-arc (v,w)
   num(w) $\leq$ max-num and w $\notin$ list imply, that w $\in S_{old}$ and w was found before the last update of max-num.
   If D(w)=True, w belongs to a handle according to induction assumption and the unmarked elements of list form the new handle.
   If D(w)=False, strong connectedness together with num(w) $\leq$ max-num ensures that a minimal path, which starts at w, cannot lead into a circle (according to proof of Lemma 18). Finiteness of the net and Lemma 17 force this minimal path to end at a node x $\in S_{list}$. num(w) $\leq$ max-num and D(w)=False imply that w $\in S_{old}$ when the last handle, which caused the update of max-num, was found. At that stage in the past of calculation, S_{list} formed a borderline and v $\in S_{new}$, i.e. v is not reachable from w without crossing S_{list}. This borderline was marked when the last handle was found. Thus at the current stage of calculation any termination node x $\in S_{list}$ reachable by eval-old(w) is marked, more formally num(x) $\leq$ max-num and D(x)=True by Lemma 15. Thus eval-old marks the minimal DFS-path and all further required handles along this DFS-path according to Lemma 18 correctly. The unmarked elements of list and the minimal DFS-path form the new handle and the marking of unmarked elements of list by mark-handle is correct.
3. Handle found with F-arc (v,w)
   An F-arc (v,w) ensures existence of a DFS-path of T-arcs $(v, a_1, a_2, \ldots, a_n, w)$.
   num(v) $\leq$ max-num implies D(v)=True by Lemma 15.
   If D(w) = True, w is marked correctly by induction assumption, eval-old(w) terminates without recursion and no nodes have to be marked.
   If D(w) = False, the recursive structure of get-handle requires that get-handle(w) has terminated and all DFS-successors of w are fully explored. Thus ll(w) is fixed. The strong connectedness of the net ensures that v is reachable from w. According to Lemma 18 eval-old(w) terminates at nodes x $\in S_{list}$. This is correct if S_{list} = { x | D(x)=True }. The recursive structure of get-handle -delete(list,x) occurs at termination of get-handle(x)- together with Lemma 11 and Lemma 12 ensure that num(v) = max{num(x) | x $\in$ list}. Thus by Lemma 15 and D(v)=True holds D(x)=True forall x $\in$ list. Thus S_{list} = { x | D(x)=True }. So eval-old marks the new handle H covering w and furthermore it marks also other handles required by the places of H.

at 2) Assume the contrary, i.e. x $\in$ P, y $\in$ T and y $\in \bullet$ x, D(x)=True and D(y)=False. There are only two points in Algorithm 10, where D(x) can be set true:

1. D(x):= True in eval-old(x)
   If y $\notin$ list, then eval-old(x) with x $\in$ P implies eval-old(y). So D(y)=True.
   If y $\in$ list, then calling eval-old from finding a handle with B- or C-arcs ensures that mark-handle marks all elements of list with num-value $>$ max-num. If num(y) $\leq$ max-num then Lemma 15 implies D(y) = True anyway. If eval-old is invoked from finding a handle with an F-arc (a,b), then a $\in$ list and num(a) $\leq$ max-num. Thus by Lemma 15 D(a)=True and as get-handle(a) is actually checked, a must be the maximal element of list. Thus num(y) $\leq$ num(a) and D(y)=True due to Lemma 15. In all cases D(y)=True contradicting the assumption.
2. D(x):=True in mark-handle
   If num(y) $\neq$ 0 then eval-old(y) occurs, because x $\in$ P.
   If num(y) = 0 a get-handle(y) occurs due to x $\in$ P. Thus y $\in$list for a while. Due to the strong connectedness of the net, get-handle(y) finds a handle and thus D(y) := True in mark-handle before update of max-num.

□

8 On the Algorithm's Time Complexity

The central idea in calculating complexity is to charge nodes and arcs for the effort they cause. This leads to constant costs per node and arc. Thus we get $O(|P| + |T| + |F|)$.

Lemma 11 ensures that there is at most one get-handle-call per node. All assignments and list handling are assigned to v. The list handling requires very restricted operations which are easily implemented in a way that they cause only constant effort. Any DFS-successor of v is reached via one arc and can cause:

1. a get-handle(w)-call
   The costs are handed over to DFS-successor w.
2. an eval-old(w)-call
   Calling eval-old(w) leads to constant costs for assignments and if-statements and costs for recursive calls of eval-old. The constant costs are assigned to the arc (v,w). It is never followed again, due to the fact that it is only followed once within get-handle when a handle is found and v gets D(v)=True. v $\in$ list or D(v) = True stops any recursive calls of eval-old in v. Lemma 13 states that any node can cause recursive eval-old-calls at most once, thus the total cost of a recursion is shared between those arcs which the recursion follows. As any arc can take part in a recursion once, its costs remain constant. Choosing a DFS-successor with minimal ll-value at a transition can be avoided by storing this DFS-successor when fixing the ll-value within get-handle.

3. a mark-handle-call
   Updates of max-num (max-num:=i) imply that max-num only increases. In mark-handle only those elements of list with num-value > max-num get marked. Thus these elements get never marked in mark-handle again. So these costs are constant per node and are assigned to the corresponding node. The costs for eval-old-calls within mark-handle are assigned according to eval-old-calls within get-handle.

Altogether any node and arc carry constant costs, which leads to $O(|P|+|T|+|F|)$.

9 Conclusions

In this paper we have shown that a particular DFS-algorithm allows to find a minimal deadlock in a strongly connected FC-net containing a given place p. The time complexity of the new algorithm is linear in the size of the net. This has positive consequences on the complexity of deciding structural liveness and boundedness of FC-nets. Embedding this new algorithm within the one given in [7] leads to the following calculation of time complexity:

check-str-connected	$O(\|P\|+\|T\|+\|F\|)$
get-minimal-deadlock (old)	$O(\|T\|(\|P\|+\|T\|+\|F\|))$
get-minimal-deadlock (new)	$O(\|P\|+\|T\|+\|F\|)$
check-s-component	$O(\|P\|\|T\|)$
matrix-rank & check-initial-marking	$O(\|P\|^2\|T\|)$

The functions get-minimal-deadlock and check-s-component are called at most $|P|$ times, all others are called at most once.

$$
\begin{aligned}
&O([|P|+|T|+|F|]+|P|[(|P|+|T|+|F|)+|P||T|]+|P|^2|T|) = \\
&O(|P||F|+|P|^2|T|) = \qquad (2|P||T| \geq |F| \text{ due to flow matrices}) \\
&O(|P|^2|T|)
\end{aligned}
$$

The time complexity is reduced from $O(|P||T||F|)$ in [7] to $O(|P|^2|T|)$ here. This moves the focus point from finding minimal deadlocks to checking them for being S-components, calculating the rank of the incidence matrix and checking the initial marking for being live. Further improvements can easily be achieved by integrating check-s-component into get-minimal-deadlock and using other techniques for checking the initial marking, see [9] or [[4], proposition 4.3]. This leads to a reduction of constant factors but does not lower the worst case time complexity since the calculation of a matrix rank requires $O(n^3)$.

The algorithm is implemented and tested for recognising LBFC-nets within the QPN-Tool, a tool for qualitative and quantitative analysis of Queueing Petri Nets, cf. [2].

Acknowledgement

The author would like to thank F. Bause, H. Beilner, B. Müller-Clostermann and four anonymous referees for their valuable comments to improve this paper.

References

1. K. Barkaoui and M. Minoux. A polynomial-time graph algorithm to decide liveness of some basic classes of bounded petri nets. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, LNCS 616, pages 62–75, Berlin Heidelberg, 1992. Springer.
2. F. Bause and P. Kemper. *QPN-Tool Version 1.0 User's Guide*. Universität Dortmund, LS Informatik 4, 1991.
3. E. Best and M.W. Shields. Some equivalence results for free choice nets and simple nets and on the periodicity of live free choice nets. In *CAAP'83, Trees in Algebra and Programming, 8th Colloquium, L'Aquila*, LNCS 159, pages 141–154, Berlin Heidelberg, 1983. Springer.
4. J. Esparza. Synthesis rules for petri nets, and how they lead to new results. In J.C.M. Baeten and J.W. Klop, editors, *Concur '90*, LNCS 458, pages 182–198, Berlin Heidelberg, 1990. Springer.
5. J. Esparza, E. Best, and M. Silva. Minimal deadlocks in free choice nets. Hildesheimer Informatikberichte 1/89, Universität Hildesheim, Institut für Informatik, 1989.
6. M.H.T. Hack. Analysis of production schemata by petri nets. Technical Report TR-94, MIT, Boston, 1972 corrected 1974.
7. P. Kemper and F. Bause. An efficient polynomial-time algorithm to decide liveness and boundedness of free-choice nets. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, LNCS 616, pages 263–278, Berlin, 1992. Springer.
8. K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. EATCS Monographs on Theoretical Computer Science. Springer, Berlin Heidelberg, 1984.
9. M. Minoux and K. Barkaoui. Polynomial algorithm for finding deadlocks, traps and other substructures relevant to Petri net analysis. Technical Report 212, Laboratoire MASI, Universite P. et M. Curie, Paris, 1988.
10. R. Tarjan. Depth-first search and linear graph algorithms. In *SIAM. J. Comp.*, volume 1, pages 146–160. 1972.