O($|P||T|$)-Algorithm to Compute a Cover of S-components in EFC-nets

Peter Kemper

Informatik IV, Universität Dortmund
44221 Dortmund, Germany

Abstract. For a strongly connected Extended-Free-Choice Net (EFC-net) N the algorithm given in this paper either finds a cover of S-components or recognizes that N is not well-formed. Well-formedness denotes the existence of a live and bounded marking for N. The algorithm is also adaptable to compute a cover of T-components. The time complexity is O($|P||T|$). According to the Rank Theorem, finding a cover of S- and T-components is useful for recognizing well-formed EFC-nets. Employing the new algorithm allows to decide well-formedness for EFC-nets in O($|P||T|^2$).

1 Introduction

In the context of Place/Transition Nets liveness and boundedness are properties which are often desired but hard to decide for a given net. For the restricted class of Extended-Free-Choice nets (EFC-nets) several algorithms are known to decide well-formedness, i.e. existence of a live and bounded marking. In [6] J. Desel mentions that the characterization of well-formed EFC-nets by the Rank Theorem can be checked by means of linear programming in polynomial time. K. Barkaoui and M. Minoux proved in [2] that liveness is decidable for bounded EFC-nets and Non-Self-Controlling Nets in $O(|P|^2|F|^2)$[1] by searching for a minimal siphon[2] which is no trap.

For FC-nets well-formedness is decidable in $O(|P|^2|T|)$, see [11, 10], according to the Rank Theorem. The algorithm is based on finding a cover of S-components and checking whether the rank of the incidence matrix has a particular value.

A cover of S-components can be found by

- exploiting the general technique of J. Ezpeleta et al. [8] based on solving a system of linear inequalities, which is not restricted to EFC-nets. This yields a generating family K of siphon/trap-components. A cover of S-components can be extracted from K in case of well-formed EFC-nets.
- following a graph oriented approach to find an S-component for any place that must be covered.

In this article we follow the graph oriented approach and this way generalize the results of [11] to EFC-nets and reduce the time complexity to find a cover

[1] P denotes the set of places, T the set of transitions and F the set of arcs.
[2] The notion of siphon is equivalent to the notion of (structural) deadlock.

of S-components by one order of magnitude, down to $O(|P||T|)$. This is possible due to a data structure which allows to find a single S-component in $O(|T|)$.

S-components and T-components reveal a detailed insight in the possible behavior of N. S- and T-components induce S- and T-invariants. By a cover of S- and T-components we provide information of an invariant for any place and transition of N and furthermore a positive S- and T-invariant covering N, which decides most of the restrictions named in the Rank Theorem as given in [6]. To decide well-formedness, it merely remains to check the rank equation, i.e. whether the number of clusters exceeds the rank of the incidence matrix by one. The time complexity of the rank detection algorithm, e.g. Singular Value Decomposition with $O(|P||T|^2)$, cf. [9], clearly dominates the time complexity of an algorithm to decide well-formedness of EFC-nets.

The article is structured as follows: Section 2 summarizes some notations and gives the theoretical background for the algorithm to compute a cover of S-components in Sect. 3. In Section 4 an example is given and analyzed in order to illustrate the algorithm before its correctness and time complexity is concerned in Sect. 5. The conclusions drawn in Sect. 6 finish the article.

2 Basic Definitions and Theorems

Definition 1 Net.
A net is a triple $N = (P, T, F)$ where P and T are non-empty, finite sets, $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$.

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. W denotes the characteristic function of F. The incidence matrix C of N is given by $c_{ij} := W(t_j, p_i) - W(p_i, t_j)$. Rank(C) denotes the rank of the incidence matrix.
3. Every non-trivial solution x $\in \mathbb{Z}^{|P|}$ of x C = 0 is an S-invariant, every non-trivial solution x $\in \mathbb{Z}^{|T|}$ of C x = 0 is a T-invariant of N, where 0 denotes the vector with all entries 0. An S-invariant x is semi-positive iff x $\neq$0 and x $\in \mathbb{N}_0^{|P|}$, it is positive iff x $\in \mathbb{N}^{|P|}$. The support of an S-invariant x is $\{p \in P | x(p) \neq 0\}$.
4. N is a

S-graph	iff $\forall t \in T$	$: \| \bullet t \| = \| t \bullet \| = 1$
T-graph	iff $\forall p \in P$	$: \| \bullet p \| = \| p \bullet \| = 1$
Free-Choice Net (FC-Net)	iff $\forall p \in P$	$: \| p \bullet \| > 1 \Rightarrow \bullet(p\bullet) = \{p\}$
Extended-Free-Choice Net (EFC-Net)	iff $\forall t, t' \in T$	$: \bullet t \cap \bullet t' \neq \emptyset \Rightarrow \bullet t = \bullet t'$

5. 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 $i \in \{1, \ldots, n-1\}$.
6. A net $N' = (P', T', F')$ is a subnet of N, $(N' \subseteq N)$, iff $P' \subseteq P, T' \subseteq T$ and $F' = F \cap ((P' \times T') \cup (T' \times P'))$.
7. $N' \subseteq N$ is an S-component (T-component) iff N' is a strongly connected S-graph (T-graph) and $T' = \bullet S' \cup S' \bullet$ $(S' = \bullet T' \cup T' \bullet)$.
8. $P' \subseteq P$ is a siphon (trap) iff $P' \neq \emptyset$ and $\bullet P \subseteq P \bullet$ $(P \bullet \subseteq \bullet P)$. A siphon (trap) is minimal iff it does not contain a siphon (trap) as a proper subset. A siphon 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.

Definition 3 Place/Transition Nets and related definitions.
A function $M : P \rightarrow \mathbb{N}$ is called marking. A Place/Transition Net is a tuple (N, M_0) where N is a net and M_0 is a marking.

1. $P' \subseteq P$ is marked at M iff $\exists p \in P' : M(p) > 0$.
2. $t \in T$ is enabled at M iff $\forall p \in \bullet t : M(p) > 0$. A transition t enabled at M can fire and thereby create a new marking M' given by $M'(p) := M(p) + W(t,p) - W(p,t)\ \forall p \in P$. This is denoted by $M[t > M'$ and M' is called reachable from M. The reachability set denoted by $R(N, M_0)$ is the reflexive and transitive closure of reachable markings from marking M_0.
3. (N, M_0) is bounded iff $\exists k \in \mathbb{N} : \forall p \in P, M \in R(N, M_0) : M(p) \leq k$.
4. (N, M_0) is live iff $\forall t \in T, M \in R(N, M_0) : \exists M' \in R(N, M) : t$ is enabled at M'.
5. N is well-formed iff $\exists M_0 : (N, M_0)$ is live and bounded.

For well-formed EFC-nets minimal siphons reveal a close relationship to traps, S-invariants and S-components.

Theorem 4 [3], Corollary 5.4, 5.5. *Let N be a well-formed EFC-net. Then*

1. *each minimal siphon of N is a trap*
2. *each minimal siphon of N is the support of an S-invariant*
3. *each minimal siphon of N generates an S-component*

Minimal siphons are very well characterized in case of EFC-nets:

Lemma 5 Characterization of Minimal Siphons. *Let N=(P,T,F) be an EFC-net. A non-empty set $D \subseteq P$ is a minimal siphon iff*

1. *$\bullet\ D \subseteq D\ \bullet$,*
2. *D is strongly connected,*
3. *for all $t \in \bullet\ D$ holds $|\bullet\ t \cap D\ | = 1$.*

Proof. follows from the general characterization in ([1],Theorem 1) and the EFC-property. The argumentation in ([7],Corollary 3.4) for FC-nets based on ([4],Lemma 6.6) can be directly extended to EFC-nets. □

Definition 6 Cluster.
Let N=(P,T,F) be an EFC-net. For every element x $\in P \cup T$, [x] denotes the smallest set containing x which includes p$\bullet$ for every place p $\in$ [x] and $\bullet$t for every transition t $\in$ [x]. [x] is called cluster of N.

The notion of cluster is used in the following characterization of well-formed EFC-nets.

Theorem 7 [5], quoted from [6], Rank Theorem.
An EFC-net is well-formed iff it has a positive S-invariant, a positive T-invariant and the number of its clusters exceeds the rank of the incidence matrix by one.

The set of clusters denote a partition of N, e.g. in the example net given in Fig. 1 the clusters are represented by shaded boxes. The EFC-property implies that every place and every transition contained inside a cluster are connected by an arc. This motivates the aggregation of net N to a cluster graph.

Definition 8 (Inverse) Cluster Graph.
Let N=(P,T,F) be an EFC-net. The cluster graph G=(V,E) is a directed graph with V := { [x] | x $\in$ P $\cup$ T} and E := { ([x],[y]) |$\exists t \in T \cap [x], p \in P \cap [y]$: $(t,p) \in F$}. G'=(V,E^{-1}) defines the inverse cluster graph.

Remark 9.
If an EFC-net N is strongly connected then the (inverse) cluster graph of N is strongly connected.

Proof. Regard an arbitrary, but fixed path (x, ..., y) in N. For any arc (p,t) with p $\in$ P and t $\in$ T in this path, [p]=[t] by definition of a cluster. For any arc (t,p) in this path ([t],[p]) $\in$ E by definition of cluster graph. Hence any y that is reachable from x in N implies that [y] is reachable from [x] in the cluster graph of N. Consequently G is strongly connected and therefore the same holds for G'. $\square$

In the sequel we want to attach a label to a net N and its cluster graph in order to describe an S-component on N.

Definition 10 Label.
A label D is a total function D: $P \cup T \rightarrow \{True, False\}$ for an EFC-net N=(P,T,F), such that $\forall t \in T : D(t) = True \Rightarrow \exists p \in \bullet t : D(p) = True$.

A node x is called labeled if D(x)=True, unlabeled otherwise.

Definition 11 Cluster Label.
A cluster label D_{cl} is a total function $D_{cl} : V \rightarrow \{True, False\}$ for a cluster graph G=(V,E) of an EFC-net N=(P,T,F) with label D, such that for [v] $\in$ V, D_{cl}([v]) := True iff $\exists$ p $\in$ P $\cap$ [v] : D(p) = True.

The notion of handle has been useful for finding siphons and S-components, cf. [7, 10, 11], and is defined here for a net and its cluster graph.

Definition 12 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 $x_i \neq x_j$, $\forall i, j \in \{1, \ldots, n-1\}, i \neq j$.

Definition 13 Cluster Handle.
Let G=(V,E) be a cluster graph of an EFC-net N with two non-empty sets $S_{cl}, S'_{cl} \subseteq V$, $S_{cl} \cup S'_{cl} = V$ and $S_{cl} \cap S'_{cl} = \emptyset$. A path $H = (x_0, x_1, \ldots, x_{n-1}, x_n)$ in G is a cluster handle iff $x_0, x_n \in S_{cl}$, $x_1, \ldots, x_{n-1} \in S'_{cl}$, $(x_i, x_{i+1}) \in E$, $\forall i \in \{0, \ldots, n-1\}$ and furthermore $x_i \neq x_j$, $\forall i, j \in \{1, \ldots, n-1\}, i \neq j$.

In the following we will only regard handles and cluster handles according to a special partition which concerns the label resp. the cluster label of a net, i.e.
$S := \{x | x \in P \cup T \wedge D(x) = True\}$, $S' := (P \cup T) \backslash S$,
$S_{cl} := \{[x] | x \in P \cup T \wedge D_{cl}([x]) = True\}$, $S'_{cl} := V \backslash S_{cl}$

Proposition 14 Existence of a Cluster Handle.
Let N=(P,T,F) be a strongly connected EFC-net with a label D, a cluster graph G and cluster label D_{cl}.
For any $t \in \bullet\, p$, $p \in P$, with $D_{cl}([t])$ = False and D(p) = True exists a cluster handle $([x_0], \ldots, [x_n])$ with $[x_{n-1}] = [t]$ and $[x_n] = [p]$.

Proof. follows directly from Remark 9 and the way cluster graph, label and cluster label are defined. □

Lemma 15 Cluster Handles Induce Handles.
Let N=(P,T,F) be an EFC-net with cluster graph G=(V,E), label D and cluster label D_{cl}. If $H_{cl} = ([x_0], \ldots, [x_n])$ is a cluster handle with $(t_{n-1}, p_n) \in F$ causing $([x_{n-1}], [x_n]) \in E$ and D(p_n)=True then

1. *a path $H = (t_0, p_1, t_1, \ldots, p_{n-1}, t_{n-1}, p_n)$ exists,*
2. *if D(t_0) = True then H is a handle, and*
3. *if D(t_0) = False then $p_0 \in [x_0]$ exists such that a handle is given by $(p_0, t_0, p_1, t_1, \ldots, p_{n-1}, t_{n-1}, p_n)$.*

Proof.

at 1) Any arc $([x_i], [x_{i+1}])$ in H_{cl} ensures by Def. 8 that an arc $(t_i, p_{i+1}) \in F$ exists. For any cluster $[x_i]$ in H_{cl} the EFC-property ensures that an arc (p_i, t_i) exists. Thus a path H exists.
at 2) From Def. 13, Def. 11 and Def. 10 directly follows that D(p_i) = D(t_i) = False holds for all p_i, t_i with i $\in \{1, \ldots, n-1\}$. Definition 13 ensures that no p_i, t_i with i $\in \{1, \ldots, n-1\}$ can occur twice in H. So if D(t_0) = D(p_n) = True then $(t_0, p_1, t_1, \ldots, p_{n-1}, t_{n-1}, p_n)$ is a handle.
at 3) By Def. 13 $D_{cl}([x_0])$ = True, thus by Def. 11 exists $p_0 \in [x_0]$ with D(p_0) = True. Hence with the same argumentation as for 1.) and 2.) we can conclude for the case D(t_0) = False that a path $(p_0, t_0, p_1, t_1, \ldots, p_{n-1}, t_{n-1}, p_n)$ exists which is a handle. □

E.g., in the net given in Fig. 1 let S={f,d} and consequently S_{cl}={[f]}. H_{cl}=([f],[b],[j],[f]) is a cluster handle and H=(d,b,a,j,h,f) is its induced handle.

By now we know that in case of strongly connected EFC-nets cluster handles exist and that they induce handles. For our main goal, the computation of a cover of S-components, it will turn out that any S-component is composed of a set of handles, which are induced by cluster handles.

3 On Finding a Cover of S-Components

Before the algorithm for finding a cover of S-components is presented we focus on the data structure it requires and on an algorithm to initialize this data structure. It is based on the inverse cluster graph $G' = (V, E^{-1})$ of an EFC-net.

The central idea of this data structure is to select a subset $\hat{E}^{-1} \subseteq E^{-1}$ of arcs from G' such that $\hat{G} = (V, \hat{E}^{-1})$ is a tree. The set of clusters V is numbered according to a preorder numbering of this tree. Assume that for any cluster [x] the maximal number of an element in the subtree with root [x] is given by rt([x]). Then the preorder numbering ensures that all elements [y] in the subtree with root [x] have numbers according to num([x]) $\leq$ num([y]) $\leq$ rt([x]). For this reason it is simple to decide if a cluster is reachable from another cluster via tree-arcs. Trivially all clusters are reachable from the root of the whole tree.

Additionally for every cluster the data structure selects one arc, called ll-arc, such that successively following these arcs leads to the root of the whole tree.

This way the data structure supports the following strategy to obtain a complete cluster handle $([x_0], \ldots, [x_n])$ for a given suffix $([x_{n-1}], [x_n])$: In general only cluster $[x_n]$ must be labeled already. We follow ll-arcs starting from $[x_{n-1}]$ until we reach a cluster [y] such that $[x_n]$ is element of the subtree with root [y]. Then we follow tree-arcs until we reach $[x_n]$. This way we find a path $([x_0] = [x_n], [x_1], \ldots, [x_{n-1}], [x_n])$. If we reach a labeled cluster $[x_i]$ while we follow ll-arcs or tree-arcs we stop because a cluster handle is already complete. This strategy yields a cluster handle if no unlabeled cluster is reached twice.

3.1 Computation of Data Structure by Depth-First-Search

The appropriate data structure is computed efficiently by adapting ideas from Tarjan's algorithm, [12]. One of the essentials there is that by recursive Depth-First-Search (DFS) it is possible to compute a tree on a graph such that all nodes are elements of the tree and nodes are numbered in preorder. All arcs of the graph are partitioned into four classes in terms of this tree.[3] Regard a dfs([v])-call which checks an arc ([v],[w]) on the inverse cluster graph G':

T(ree)-arcs belong to the tree. In terms of DFS, this type of arc leads to a node which has not been reached before.
Formal criterion : num([w]) = 0.

[3] For more detailed descriptions of DFS-algorithms in this context see [10, 11].

F(orward)-arcs bypass a path of T-arcs, i.e. they lead from a node [v] to a node [w] in the subtree with root [v]. In terms of DFS, this type of arc leads to a node which was already visited and which is reachable via a 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]).

B(ack)-arcs are reverse to F-arcs, i.e. their starting node [v] is element of the subtree with root [w]. In terms of DFS, this type of 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] $\in$ path (root,. . .,[v]) of T-arcs.

C(ross)-arcs lead to a node in a different subtree. The nodes of this subtree have lower numbers than [v] due to preorder numbering and construction by DFS. In terms of DFS, this type of arc leads to a node [w] whose dfs([w])-call terminated before [v] was found. This means that no path of T-arcs from [w] to [v] exists.
Formal criterion: num([w]) $<$ num([v]) $\wedge$ [w] $\notin$ path (root,. . .,[v]) of T-arcs.

The formal criteria are found in Algorithm 17. The major advantage of the classification of arcs is that any circle contains at least one B-arc. The following value characterizes certain paths which end with a B-arc. The definition is adapted from [10] and differs from the one in [12] concerning C-arcs.

Definition 16 low-link value (ll-value).
Let $G' = (V, E^{-1})$ be a strongly connected, inverse cluster graph with a partition of E^{-1} into T-,F-,C-, and B-arcs and a numbering of V according to preorder in the tree of T-arcs. The low-link value of any node [x] $\in$ V is defined by:

ll([x]) := min({ num([x]) } $\cup$ { num([z]) | ([x],[z]) is a B-arc }
$\cup$ { ll([y]) | ([x],[y]) is a T-arc or C-arc })

An F-arc ([x],[y]) cannot be relevant for ll([x]) because there always exists a path ([x],. . .,[y]) of T-arcs that would contribute to ll([x]) if ll([y]) was really minimal.

Algorithm 17 performs Depth-First-Search to fill this data structure:

for every x $\in$ P $\cup$ T: [x] gives the cluster that node x belongs to.
for every [x] $\in$ V:

- **num([x])** is a unique number to identify [x] according to preorder in the tree of T-arcs.
- **rt([x])** gives the maximal num-value of all clusters that belong to the subtree with root [x]. Thus any cluster [y] which is element of the subtree with root [x] is characterized by num([x]) $\leq$ num([y]) $\leq$ rt([x]).
- **successor-list([x])** lists the arcs of N which cause T-arcs on G' that start in [x], i.e. it implicitly gives the list of sons in the tree of T-arcs.
- **ll([x])** according to Def. 16 for G'.
- **ll-arc([x])** gives the arc of N which leads to the cluster where ll([x]) got its value from.

Algorithm 17. Initialisation of Data Structure

```
Input: N = (P,T,F) strongly connected EFC-net with r ∈ P
Program:
begin
    forall p ∈ P                        { create cluster data structure for all clusters }
        if [p] = null
        then c :=create-cluster() ;
            num(c)=rt(c)=ll(c):=0 ; path(c):=False ; successor-list(c):=empty ;
            forall t ∈ p • : [t] := c ;
            choose t ∈ p •;
            forall p' ∈ • t : [p'] := c ;
    i := 0 ;
    dfs( r ) ;                          { start DFS with root cluster [r] }
end

Function: dfs( p )
begin
    i := i + 1 ; num([p]) := i ; ll([p]) : = i ; path([p]) : = True ;
    forall t ∈ • p                                  { regard inverse arcs }
    begin
        if num([t]) = 0                             {  T-arc ([p],[t]) }
        then
            append( successor-list([p]) , (p,t) ) ;
            forall p' ∈ •t                          { forall p' ∈ [t],[t] ≠ [p] }
            begin
                dfs( p' ) ;
                if ll([t]) < ll([p])
                then                                {  update ll-value, ll-arc }
                    ll-arc([p]) := (p,t) ; ll([p]) := ll([t]) ;
            end
        if num([t]) > num([p])                      {  F-arc ([p],[t]) }
        then ignore
        if ( 0 < num([t]) < num([p]) ∧ path([t])=True
        then                                        { B-arc ([p],[t]) }
            if num([t]) < ll([p])
            then                                    {  update ll-value, ll-arc }
                ll-arc([p]) := (p,t) ; ll([p]) := num([t]) ;
        if ( 0 < num([t]) < num([p]) ∧ path([t])=False
        then                                        { C-arc ([p],[t]) }
            if ll([t]) < ll([p])
            then                                    {  update ll-value, ll-arc }
                ll-arc([p]) := (p,t) ; ll([p]) := ll([t]) ;
    end
    path([p]) := False ; rt([p]) := i ;
end
```

One way to implement the cluster data structure is to give every node a pointer which points to the cluster it belongs to. As a result an update of any information concerning a cluster is possible in constant time and does not depend on the number of nodes that belong to the cluster.

A merely temporary data structure used only internally in Algorithm 17 is path([x]) which denotes if [x] belongs to the recursion stack of dfs, i.e. path([x])=True for all [x] with dfs(x) not finished yet. This represents the path (root,...,[v]) on the tree of T-arcs with dfs(v) on top of the recursion stack.

The algorithm is a variation of DFS, so we do not prove the obvious and just summarize its properties:

1. All clusters are visited and all inverse cluster arcs are classified in terms of T-, C-, B- and F-arcs.
2. The set of T-arcs describes a tree with root [r] which contains all clusters. The successor-list of a cluster does not contain multiple arcs connecting it to another cluster. The num-value gives the number according to a preorder visit to the nodes of the tree of T-arcs.
3. A B-arc closes a circle on this tree.
4. A C-arc connects two different subtrees from higher num-values to lower num-values.
5. An F-arc leads to a cluster which is in the same subtree and is reachable via a path of T-arcs.
6. A cluster [y] is reachable via a path of T-arcs from a cluster [x] iff num([x]) $\leq$ num([y]) $\leq$ rt([x]).
7. The algorithm reflects Def. 16 concerning ll-values. All clusters, except for the root cluster [r], have exactly one ll-arc.
8. rt is set correctly, as the variable i counts the number of clusters reached so far and thus gives the appropriate value for rt([p]) if dfs(p) terminates. If further p' $\in$ [p] are regarded, the p' which is regarded last determines the correct value of rt([p]).
9. The time complexity is O($|P| + |T| + |F|$) because for all places p dfs(p) is called exactly once, which causes constant effort per arcs (t,p) for all t $\in \bullet$ p and (p',t) for all p' $\in \bullet$ t.

Figure 2 gives a graph representation of the data structure computed for the example net in Fig. 1 with root cluster [b]. Elements of successor-lists are represented by solid arcs, ll-arcs by dashed arcs. Note that the choice of arcs depends on the order they are checked by DFS.
A useful property of the low-link value is:

Lemma 18. *Let G'=(V,E^{-1}) be the inverse cluster graph of a strongly connected EFC-net with a data structure computed by Algorithm 17 with root cluster [r]. Then for any [x] $\in$ V holds: ll([x])=num([x]) $\Rightarrow$[x]=[r]*

Proof. Assume the contrary: [x] $\neq$ [r] and ll([x]) = num([x])
Regard a path MP=($[x_0], \ldots, [x_i], \ldots, [x_n]$) on G' with $[x_0] = [r]$,$[x_i] = [x]$ and

$[x_n] = [r]$ which consists of T-arcs on the part $[x_0], \ldots, [x_i]$ and with minimal number of B-arcs.
Such a path, which is a circle, exists because
1. num([r]) = 1 is minimal, so num([r])<num([x]) and [x] is reachable via a path of T-arcs from [r], and
2. due to Remark 9 the graph is strongly connected, hence at least one path leading from [x] to [r] exists and therefore a path with a minimal number of B-arcs can be selected.
The part $[x_0], \ldots, [x_i]$ of MP consists of T-arcs, consequently it contains no B-arc. So the first B-arc $([x_{j-1}], [x_j])$ in MP fulfills $j > i$.
If $[x_j] = [x]$ the path MP would not be the one with minimal number of B-arcs, because path MP'=$([x_0], \ldots, [x_i], [x_{j+1}], \ldots, [x_n])$ would fulfill the above restrictions for MP as well and has one B-arc less.
For this reason $[x_j] \neq [x]$. Together with num([x])=ll([x]) and ll([x]) $\leq$ num($[x_j]$) (by definition of ll-value and existence of path [x] to $[x_j]$) we obtain num([x]) < num($[x_j]$). Regard two cases:
1. num([x]) < num($[x_j]$), but num($[x_j]$) > rt([x])
The path $[x_i], \ldots, [x_{j-1}]$ consists of T-,C-, and F-arcs. T- and F-arcs remain in the same subtree, therefore their rt-value is $\leq$ rt($[x_i]$). C-arcs lead to subtrees with lower num-values and rt-values < rt($[x_i]$). Altogether num($[x_{j-1}]$) $\leq$ rt($[x_i]$) = rt([x]) holds. Arc $([x_{j-1}], [x_j])$ is a B-arc, which implies by definition of B-arcs that num($[x_j]$) < num($[x_{j-1}]$). Hence num($[x_j]$) < num($[x_{j-1}]$) $\leq$ rt([x]) which contradicts the assumption above.
2. num([x]) < num($[x_j]$) $\leq$ rt([x])
Consequently $[x_j]$ is reachable via T-arcs from $[x_i]$ which contradicts the minimality of MP. Thus there is no such B-arc $([x_{j-1}], [x_j])$.
A direct consequence of the classification of arcs is that any arc leading to [r] is classified as a B-arc, hence MP contains at least the B-arc $([x_{n-1}], [r])$. Therefore assumption num([x])=ll([x]) for [x] $\neq$ [r] was false. □

3.2 Algorithm for Finding a Cover of S-Components

The computation of a cover is based on the computation of single S-components. Theorem 4 implies that finding a minimal siphon is useful for finding an S-component in case of a well-formed EFC-net N. If a minimal siphon does not turn out to be an S-component, then N cannot be well-formed.

In FC-nets the computation of minimal siphons is possible by a successive computation of handles to cover all input transitions of any place that belongs to a minimal siphon. This idea has been successfully used for algorithms for FC-nets, [10, 11], and originates from [7].

For EFC-nets the notion of handle is not sufficient to ensure $|\bullet t \cap D| = 1$, $\forall t \in \bullet D$ for the minimal siphon $\mathcal{D}$. If we use handles that are induced by cluster handles according to Lemma 15 we gain the desired effect.

The algorithm can be described informally as follows: As long as an uncovered place exists, compute an S-component (function mark-s-component) to cover this place and thereby reduce the set of uncovered places. Compute an S-component by attaching a label D to N such that a minimal siphon and its preset is labeled. This is done by successively labeling cluster handles $([x_0], \dots, [x_n])$ for labeled p $\in [x_n]$ and an unlabeled t $\in [x_{n-1}] \cap \bullet p$ and its corresponding handle $(p_0, t_0, \dots, t_{n-1}, p_n)$ with $t_{n-1} = t$ and $p_n = p$ (function mark-handle). This way it can be ensured that label D actually labels a minimal siphon and its preset. For such a minimal siphon it has to be checked whether it is a trap, i.e. its postset is labeled also. If this condition is not satisfied, the EFC-net cannot be well-formed. If satisfied, it remains to check whether the minimal siphon is an S-component, i.e. if any transition contains at most one output place which is element of the siphon. As a shortcut this is checked during computation of the minimal siphon, because it becomes obvious if a t_0 of a corresponding handle is labeled already.

The main idea in following the cluster handle $([x_0], \dots, [x_n])$ is to start at $[x_{n-1}]$ and proceed backwards on the net N, i.e. forward on the inverse cluster graph, heading towards $[x_n]$ as a suitable $[x_0]$. In order to avoid circles we follow ll-arcs until a cluster is reached, such that $[x_n]$ is element of the subtree of T-arcs, then we follow a path on the tree down to $[x_n]$. Naturally on this way we stop as soon as a labeled cluster is reached. In the worst case we stop at $[x_n]$ which is labeled, because we originally wanted to get a handle for an unlabeled input transition of a labeled p $\in [x_n]$. It remains to show that this strategy avoids circles and ensures $| \bullet t \cap D| = 1, \forall t \in \bullet D$ which is required by Lemma 5.

Algorithm 19. Cover of S-components

Input: N=(P,T,F) strongly connected EFC-Net
Output: Either a cover of S-components or message: N is not well-formed
Initialisation: init-data-structure(N, r) by Algorithm 17 with arbitrary r $\in$ P
stack := empty; Uncovered := P; Cover := $\emptyset$;

```
Program:
begin
    i := 0 ;                          { index for S-component SC_i }
    while exists p ∈ Uncovered
    begin
    forall x in P ∪ T                 { reset label D and cluster label DCL }
        D(x) := False ; DCL([x]) := False ;
    mark-s-component(p);              { label S-component containing p }
    SC_i := {x|x ∈ P ∧ D(x) = True}
    Cover := Cover ∪{SC_i};
    Uncovered := Uncovered \SC_i ;
    i := i+1 ;
    end
end
```

```
Function: mark-s-component(p)
begin
    D(p) := True ; DCL([p]) := True ;
    choose t ∈ • p ;
    mark-handle( t, p ) ;              { compute first handle }
    push( stack, ( t, p) ) ;
    while stack not empty
    begin                              { compute successively additional handles }
        pop( stack, (t', p')) ;
        forall t ∈ • p', t ≠ t'
            mark-handle( t, p' ) ;
    end
    forall t ∈ T, D(t)=False           { test trap property }
        if DCL([t])= True
        then Stop: Not well-formed!
end

Function: mark-handle( t_{n-1}, p_n )
begin
    finish := False ; t_i := t_{n-1} ;
    while finish = False
    begin
        if D(t_i) = True                    { t_i has two labeled output places }
        then Stop: Not Well-formed!
        D(t_i) := True ;
        if DCL([t_i]) = True
        then finish := True ;                        { handle complete }
        else (t_{i-1}, p_i) := get-next-pair( t_i, p_n ) ;
            D(p_i) := True ; DCL([t_i]) := True ;    { [t_i] = [p_i] }
            push( stack, (t_{i-1}, p_i)) ;
            t_i := t_{i-1} ;                        { continue with next cluster }
    end
end

Function: get-next-pair( t_i, p_0 )
begin                                   { follows ll-arc or suitable T-arc }
    if num([t_i]) ≤ num([p_0]) ≤ rt([t_i])
    then (p_i, t_{i-1}) := find-tree-successor( t_i, p_0 ) ;
    else (p_i, t_{i-1}) := ll-arc([t_i]) ;
    return (t_{i-1}, p_i) ;
end
```

Function: find-tree-successor(t_i, p_0)
finds (p_i, t_{i-1}) from successor-list($[t_i]$),
such that num($[t_{i-1}]$) $\leq$ num($[p_0]$) $\leq$ rt($[t_{i-1}]$)

Note that D is a label and DCL a cluster label, i.e. according to Def. 10 and Def. 11 no transition is labeled without a labeled input place and no cluster is labeled without labeling one of its places. This is satisfied in mark-s-component(p) which labels p and [p]. In mark-handle(t_{n-1}, p_n) transition $t_{n-1} = t_i$ gets labeled with the consequence that either $[t_i]$ is labeled already, which implies that it contains a labeled place, or an appropriate $p_i \in [t_i]$ is computed by get-next-pair(t_i, p_n) which gets labeled together with $[t_i] = [p_i]$. Thus Algorithm 19 concerns the definition of label and cluster label.

4 Example

In this section we want to regard an example to illustrate Algorithm 19. For the strongly connected EFC-net given in Fig. 1 Algorithm 17 computes the data structure shown in Fig. 2. Solid arcs and their inscriptions give the elements of the successor-list of a cluster, dashed arcs represent ll-arcs. For cluster {d,e,f,g} it is possible to exchange (f,h) in successor-list for (f,i) or (g,h). The selection depends on which arc is checked first.

With the help of this data structure we want to find S-components for places f and c. In order to simplify the notation we identify clusters by their numbers. Let S:={ x | D(x) = True } and S_{cl} := { num([x]) | DCL([x]) =True }.

4.1 mark-s-component(f)

S = {f}, S_{cl} = {2} and stack =empty when mark-handle(h,f) is called. In mark-handle num([h]) = 3 $\not\leq$ num([f]) = 2 $\leq$ rt([h]) = 3 forces the algorithm to follow ll-arc (j,a). At cluster [a] it follows T-arc (b,d) because num([a]) = 1 $\leq$ num([f]) = 2 $\leq$ rt([a]) = 9 and cluster [d] is the element of successor-list([a]) that fulfills num([d]) = 2 $\leq$ num([f]) = 2 $\leq$ rt([d]) = 4. Cluster [d] is labeled, thus a complete cluster handle is found. During this computation several nodes and clusters were labeled: S = {f,h,j,a,b,d}, S_{cl} = {2,3,1}. The cluster handle is (2,1,3,2) and its induced handle is (f,d,b,a,j,h,f). The stack contains (a,j),(d,b),(h,f) by now. (h,f) is popped and mark-handle(i,f) terminates right away after labeling transition i because cluster [i] is labeled already. This is correct because [i] contains the labeled place j and this way handle (j,i,f) is found and labeled.

This reveals the advantages of Definitions 10 and 11, because DCL([i]) signals if a labeled place in [i] exists. Hence it is not necessary to search for the labeled place and it ensures that any cluster contains at most one labeled place.

(d,b) is popped from the stack and mark-handle(m,b) follows ll-arcs (n,o) and (p,e) until the labeled cluster [e] is reached. This way S = {f,h,j,a,b,d,i,m,n,o,p,e}, S_{cl} = {2,3,1,5,6} and the cluster handle (2,6,5,1) with its induced handle (f,e,p,o,n,m,b) is found and labeled.

By now the stack contains (a,j),(o,n) and (e,p) which are all popped without causing any further mark-handle calls. As a result the minimal siphon (f,j,b,n,p)

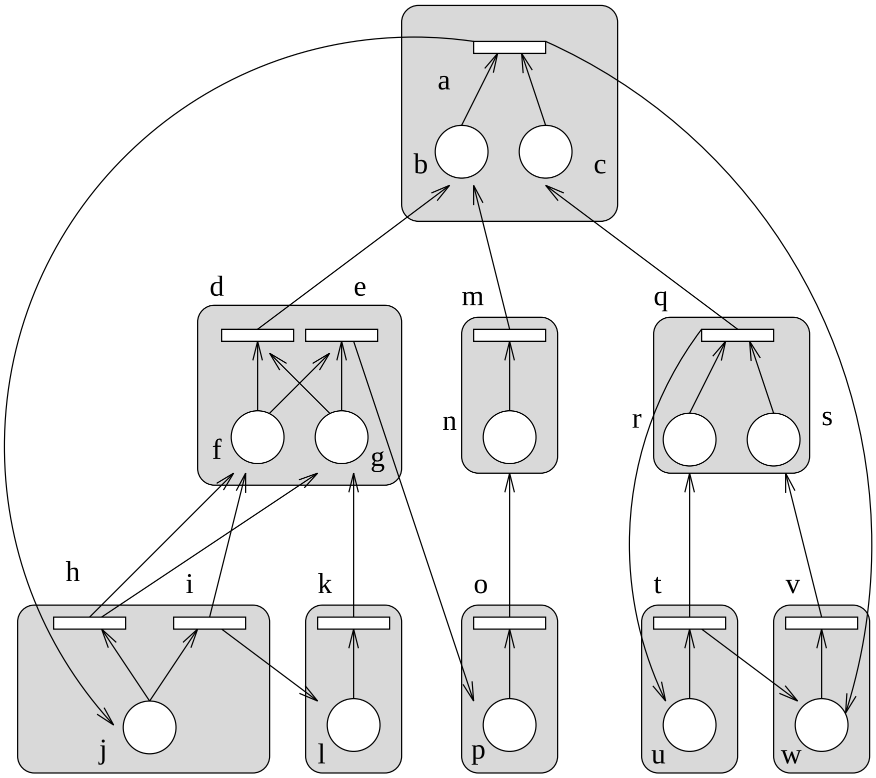

Fig. 1. A strongly connected EFC-net with shaded clusters

is found. It remains to test whether a labeled cluster exists that contains an unlabeled transition. This is not the case, hence the minimal siphon is a trap. The situation that a transition was labeled a second time did not occur, so any labeled transition has exactly one labeled input and output place. Hence the minimal siphon generates an S-component and SCi={f,j,b,n,p}.

4.2 mark-s-component(c)

All labels are set back to False ahead of mark-s-component(c). S = {c}, S_{cl} = {1} and stack =empty when mark-handle(q,c) is called. In mark-handle the algorithm follows ll-arc (s,v) due to num([q]) = 7 $\not\leq$ num([c]) = 1 $\leq$ rt([q]) = 9. Note that the way an ll-arc is chosen for a cluster avoids getting trapped in a circle. In this situation the algorithm avoids getting trapped in the circle between cluster [t] and [q]. This was realized at the initialization of the data structure by ll([v]) = 1 < ll([t]) = 7 and (s,v) was selected for ll-arc([q]).

From cluster [v] the algorithm follows ll-arc (w,a) and reaches the labeled cluster [a]. S = {c,q,s,v,w,a}, S_{cl} = {1,7,9} and the cluster handle (1,9,7,1) and its induced handle (c,a,w,v,s,q,c) are found. The stack contains (v,s),(a,w), and

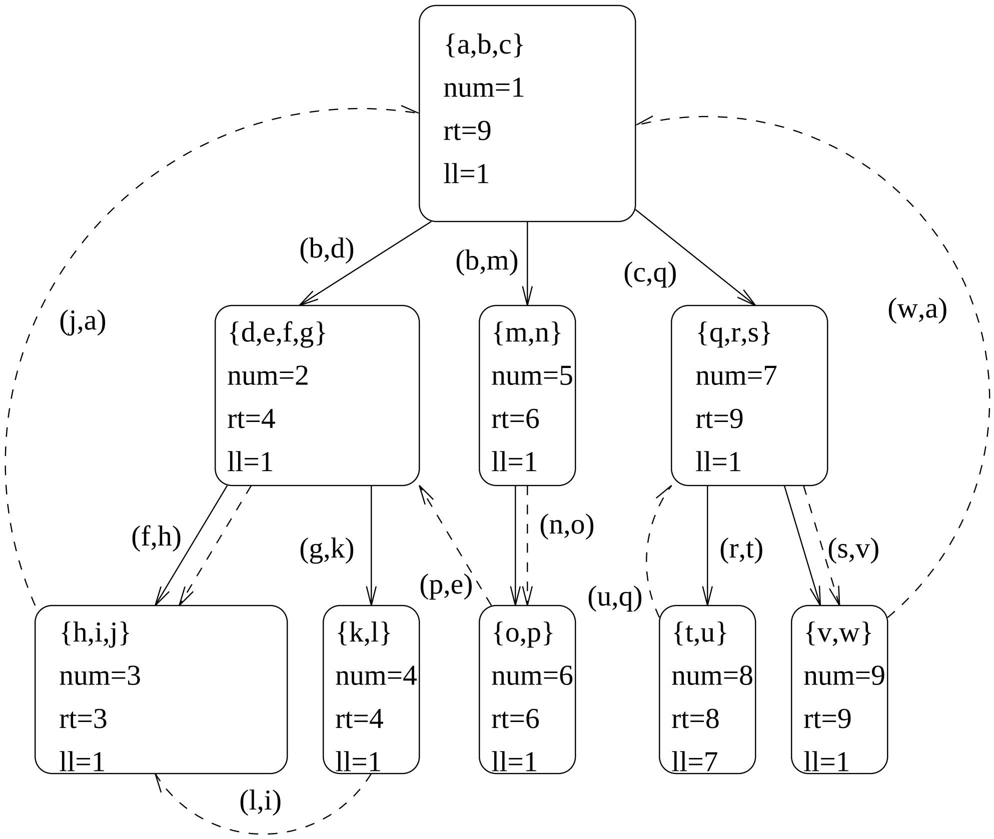

Fig. 2. Corresponding data structure based on the inverse cluster graph

(q,c). (q,c) is popped. (a,w) is popped and mark-handle(t,w) follows ll-arc (u,q). S = {c,q,s,v,w,a,t,u}, S_{cl} = {1,7,9,8} but transition q is labeled already when reached via ll-arc(u,q). The algorithm stops with message: N is not well-formed. This is appropriate because the minimal siphon found by successively adding handles cannot be an S-component due to $q \bullet \cap S = \{c, u\}$, i.e. transition q has two output places that would belong to the S-component. Hence the example net is not well-formed. For example, a marking with one token on each of the places b, r, and s is live and unbounded.

5 Correctness of Algorithm 19

5.1 Termination

Theorem 20. *Algorithm 19 terminates for any finite strongly connected EFC-net N.*

Proof.

1. Within mark-handle any visited node x either is labeled D(x)=True and the function terminates or x gets labeled D(x)=True. The same holds for [x] and DCL([x]). In mark-handle no D(x) or DCL([x]) is set back to False again, hence due to the finiteness of N mark-handle stops at a labeled node or cluster.
2. Within the "forall t $\in \bullet$p, t $\neq$ t' "-loop in mark-s-component no D(x) is set back to False, for this reason the mark-handle calls terminate and the forall-loop terminates.
3. In consequence the "while stack not empty"-loop terminates because any node can at most once be pushed on the stack within mark-handle.
4. The "forall t $\in$ T, D(t)=False"-loop in mark-s-component obviously terminates, because it just checks a special condition to stop the whole algorithm. Altogether mark-s-component terminates.
5. The "while exists p $\in$ Uncovered"-loop in program terminates because SC_i at least contains the chosen node p. Therefore Uncovered looses at least one element per iteration. As Uncovered is filled only initially the finiteness of N ensures that the while-loop terminates.

Altogether the program terminates either with message N not well-formed or a set Cover which contains a selection of subsets of P. $\square$

5.2 Function mark-handle Labels a Cluster Handle

Lemma 21. *Let $S = \{ [x] \mid [x] \in V \wedge DCL([x])=True \}$ and S'=V$\setminus$ S be fixed before a mark-handle-call. A mark-handle-call(t,s) for s $\in$ P, D(s) =True and t $\in \bullet$ s labels a cluster handle $([x_0], \ldots, [x_n])$ with s $\in [x_n]$ and t $\in [x_{n-1}]$, i.e. at termination of mark-handle for all $[x_i]$ with $0 < i < n$ holds $DCL([x_i])$=True.*

Proof. Mark-handle follows a path $([x_n], [x_{n-1}], \ldots, [x_0])$ on the inverse cluster graph starting at t $\in [x_{n-1}]$ heading towards s. The path stops at the first labeled cluster $[x_0]$ which is reached. If $[x_0] \in$ S the inverse path describes a cluster handle, because any cluster reached by mark-handle is labeled and mark-handle stops when it reaches a labeled cluster. If $[x_0] \notin$ S, it was labeled within mark-handle and the inverse path does not describe a cluster handle. In order to show that this cannot happen assume that $[x_0]$ is labeled within mark-handle.

Then the path followed on the inverse cluster graph ends with a circle $([x_j], \ldots, [x_1], [x_0])$, i.e. $[x_j] = [x_0]$.

The way find-tree-successor(t_i, p_0) selects a successor cluster $[t_{i-1}]$ ensures, that for the successor cluster the condition num($[t_i]$) $\leq$ num($[t_{i-1}]$) $\leq$ num($[p_0]$) $\leq$ rt($[t_{i-1}]$) $\leq$ rt($[t_i]$) will hold. So if this condition holds for at least one cluster of the circle it will hold for the last cluster $[x_0]$ as well. Hence it holds when starting the circle at cluster $[x_j](= [x_0])$ which implies that $([x_j], \ldots, [x_1], [x_0])$ contains only T-arcs. Hence it is no circle.

As a result this condition cannot hold for any cluster in the path and all successors must be determined by ll-arc($[t_i]$).

Hence for all arcs $([x_i],[x_{i-1}])$ holds (p_i,t_{i-1})=ll-arc($[x_i]$) with $t_{i-1} \in [x_{i-1}]$. Assume $([x_i],[x_{i-1}])$ is an arc of the following type:

T-arc by definition of ll-value is ll($[x_i]$)=ll($[x_{i-1}]$) because $[x_{i-1}]$ caused the update of ll-arc($[x_i]$).

C-arc same as for T-arcs

F-arc are ignored because a path of T-arcs exists to substitute an F-arc. Algorithm 17 does not concern F-arcs for an update of ll-arc.

B-arc by definition ll($[x_i]$)=num($[x_{i-1}]$). If $[x_{i-1}] \neq$ [r], Lemma 18 ensures that ll($[x_{i-1}]$) $<$ num($[x_{i-1}]$). $[x_{i-1}]$=[r] cannot occur because root cluster [r] fulfills the condition for following tree-successors which was excluded above.

Altogether the sequence of ll-values is decreasing at B-arcs and constant at T- and C-arcs. As the circle contains at least one B-arc, ll($[x_j]$) $>$ ll($[x_0]$) holds which contradicts $[x_j]=[x_0]$. □

Lemma 21 ensures that mark-handle follows a cluster handle. From function mark-handle it is obvious that the appropriate clusters are labeled. Lemma 15 ensures that the cluster handle induces a handle on N. The elements of this handle are selected in mark-handle by employing function get-next-pair. It is easy to see that these nodes get labeled also.

5.3 Set SC_i is a Minimal Siphon

The label D labels a minimal siphon $\mathcal{D}$ and its preset $\bullet\mathcal{D}$:

1. It is a siphon, i.e. $\bullet\mathcal{D} \subseteq \mathcal{D}\bullet$
   Assume the contrary: after termination of mark-s-component(p) exists $\hat{t} \in \bullet\hat{p}$ for $\hat{p} \in P$ with D($\hat{p}$)=True and D($\hat{t}$)=False.
   It is easy to see that mark-handle($\hat{t}$,$\hat{p}$) causes D($\hat{t}$)=True as a contradiction. Hence it remains to show that mark-handle($\hat{t}$,$\hat{p}$) occurs or $\hat{t}$ gets labeled anyway.
   Case $\hat{p}=p$
   if $\hat{t}$ is chosen as t then mark-handle($\hat{t}$,$\hat{p}$) is called in the next step of mark-s-component(p). Otherwise mark-handle($\hat{t}$,$\hat{p}$) is called within the forall-loop in mark-s-component(p), because ($\hat{p}$, t) is pushed on the stack.
   Case $\hat{p} \neq p$
   In this case D($\hat{p}$):=True can only occur in mark-handle for a p'=$\hat{p}$. If $\hat{t}$=t' it is labeled in the next iteration of the while-loop in mark-handle. Otherwise $\hat{t} \neq$ t' and push(stack, (p',t')) ensures that $\hat{p}$ becomes an element of the stack. Therefore within the while-loop of mark-s-component(p) apparently element ($\hat{p}$,t') is popped and due to $\hat{t} \neq$ t' mark-handle($\hat{t}$,$\hat{p}$) occurs. Thus D($\hat{t}$)=True contradicting the assumption and the labeled subgraph describes a siphon.
2. It is strongly connected.
   At the first call of mark-handle, only one cluster is labeled, for this reason a circle is computed, which is strongly connected. Adding further handles which start and end in this subgraph cannot destroy strong connectedness.

3. $|\bullet t \cap \mathcal{D}| = 1$
This condition is ensured by searching unlabeled clusters. Labeling a cluster results in labeling exactly one (!) of its places. If a cluster is labeled already, only additional transitions of this cluster become labeled by mark-handle. Consequently the condition is satisfied.

Altogether $\mathcal{D}$ is a minimal siphon according to Lemma 5.

5.4 Set SC_i generates an S-component

The minimal siphon $\mathcal{D}$ must generate an S-component according to Theorem 4.

It has to be checked whether $\bullet\mathcal{D} = \mathcal{D}\bullet$, i.e. $\mathcal{D}$ is a trap. This is done by the forall-loop in mark-s-component which looks for unlabeled transitions in labeled clusters. If an unlabeled transition t with a labeled cluster [t] is found, then t $\in \mathcal{D}\bullet$, $t \notin \bullet\mathcal{D}$. So $\mathcal{D}$ is not a trap. According to Theorem 4 the net is not well-formed, i.e. the message is correct. Otherwise $\bullet\mathcal{D} = \mathcal{D}\bullet$.

As siphon $\mathcal{D}$ is strongly connected, $\bullet\mathcal{D} = \mathcal{D}\bullet$ and $|\bullet t \cap \mathcal{D}| = 1$ is ensured by labeling exactly one place per labeled cluster, it remains to check $|t \bullet \cap \mathcal{D}| = 1$ to verify the S-component-property. This is tested whenever a labeled cluster is reached, causing the algorithm to stop with the appropriate message. Reaching a labeled cluster is the only possibility to label a transition twice, because all input transitions of places are explicitly labeled by calling mark-handle. For this reason the condition is sufficiently tested in mark-handle.

5.5 The Set of S-components is a Cover

Within the main program an S-component is computed only for a place which is uncovered so far. This results in a set Cover $= \{SC_1, \ldots, SC_n\}$ which is obviously a cover but not necessarily a minimal cover, because all places of an S-component SC_i can be covered be the set of S-components $\{SC_{i+1}, \ldots, SC_n\}$.

5.6 Time Complexity

The time complexity summarizes the effort for initialization of the data structure which is O($|P| + |T| + |F|$) and the effort for finding the set of S-components. Every call of mark-s-component is necessary to cover a place which is uncovered so far. This implies that the set of S-invariants induced by the S-components is linearly independent. Hence a cover contains at most $k \leq |P| -$rank(C)$\leq |P|$ S-components.

The calculation of time complexity exploits two simple observations to gain the effort for a single S-component:

- Any S-component contains at most $2|T|$ arcs. In this case a computation which does only concern arcs that belong to a single S-component is in O($|T|$) provided the effort per arc is constant.

– Any tree with $|V|$ nodes contains $|V|-1$ arcs. If for any node of the tree a successor is computed at most once, the overall search effort is in O($|V|$). This is the case for searching a successor by find-tree-successor on the tree of T-arcs of G'.

The effort for finding an S-component contains two parts: the effort for the "forall t $\in$ T, D(t)=False"-loop which is obviously O($|T|$) and the effort for several mark-handle(t,p') calls.

We want to calculate the effort for all get-next-pair calls in the computation of a single S-component separately:

Due to the fact that in mark-handle get-next-pair is only called for unlabeled clusters $[t_i]$, which get labeled in the next step of mark-handle, at most one get-next-pair call per cluster occurs in the computation of a single S-component. Function get-next-pair either returns the corresponding value of ll-arc with a constant effort or it uses find-tree-successor to get the appropriate successor from the tree of T-arcs of the inverse cluster graph. Hence any T-arc in this tree is checked at most once over all get-next-pair calls. So the effort of all get-next-pair calls is in $O(|V|-1)$, because the tree of T-arcs has $|V|$ nodes and $|V|-1$ arcs. By $|V| \leq$min$(|P|,|T|) \leq |T|$ we yield O($|T|$).

It remains to calculate the effort for several mark-handle(t,p') calls without counting the effort for get-next-pair. Note that for any arc (t,p') $\in$ F mark-handle(t,p') is called at most once. Therefore the constant part of a single call can be assigned to its corresponding arc (t,p'). The algorithm follows a path and regards pairs (p',t') at most once, because all p' and t' become labeled and the while-loop terminates when a labeled t or [t] is reached. Hence only constant effort arises for any pair (p',t') which shall be assigned to the corresponding arc. All arcs, which are checked and effort is assigned to, actually belong to the S-component which is computed. Any S-component contains at most $2|T|$ arcs which causes the overall effort neglecting get-next-pair to be O($|T|$). If the minimal siphon is not an S-component, mark-handle stops at the first arc that exceeds the number of arcs an S-component can contain, hence the effort remains on O($|T|$) also.

Altogether a single S-component is computed in O($|T|+|T|+2|T|$)=O($|T|$). For $k<|P|$ S-components we finally get O($|P|+|T|+|F|+k|T|$) = O($|P||T|$) $\leq$ O(n^2) for $n = max\{|P|,|T|\}$. Note that resetting label D and cluster label DCL for each S-component can be avoided by using numbers instead of a boolean label.

6 Conclusions

The algorithm with O($|P||T|$) time complexity presented in this paper is suitable for different purposes.

1. If a well-formed EFC-net is given, the algorithm computes a cover of S-components.

2. If a strongly connected EFC-net N is given, the algorithm supports the decision whether N is well-formed or not. It either finds a minimal siphon which generates no S-component and hence N is not well-formed, or it finds a cover of S-components.
3. The duality theorem for well-formed EFC-nets in [6] states that the dual net $N^d = (T, P, F^{-1})$ of a well-formed EFC-net N is a well-formed EFC-net. It follows straightforwardly that an algorithm to compute a cover of S-components can be used to compute a cover of T-components as well, because S-components of N are T-components of N^d. Consequently by applying Algorithm 19 on N^d we get a cover of T-components of N. It is easy to formulate the algorithm in order to compute T-components directly. The time complexity remains the same.

Any S-component (T-component) of an EFC-net induces a semi-positive S-invariant (T-invariant). Hence a cover of S-components (T-components) induces a positive S-invariant (T-invariant) which covers N. For recognizing well-formed EFC-nets by the Rank Theorem it just remains to check whether the number of clusters exceeds the rank of the incidence matrix by one. As far as we know, computing the rank of a singular $(|P|\text{x}|T|)$-matrix takes $\text{O}(|P||T|^2)$, cf. [9], and therefore clearly dominates the time complexity for an algorithm to recognize well-formed EFC-nets based on the Rank Theorem.

Acknowledgement

The author would like to thank Prof. H. Beilner and Dr. F. Bause for their valuable comments to improve this paper.

References

1. K. Barkaoui and B. Lemaire. An effective characterization of minimal deadlocks and traps in Petri nets based on graph theory. In *Application and Theory of Petri Nets*, Berlin, 1989. Springer.
2. 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.
3. E. Best and J. Desel. Partial order behaviour and structure of petri nets. In *Formal aspects of computing*, volume 2(2), pages 123–138. 1990.
4. P.S. Best, E.and Thiagarajan. Some classes of live and safe Petri nets. In K. Voss, H.J. Genrich, and G. Rozenberg, editors, *Concurrency and Nets, Advances of Petri Nets*, pages 71–94, Berlin, 1987. Springer.
5. J. Campos, G. Chiola, and M. Silva. Properties and performance bounds for closed free choice synchronized monoclass queueing networks. *IEEE Transactions on Automatic Control*, AC-36(12):1368–1382, 1991.
6. J. Desel. A proof of the rank theorem for extended free choice nets. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, LNCS 616, pages 134–153, Berlin, 1992. Springer.
7. J. Esparza, E. Best, and M. Silva. Minimal deadlocks in free choice nets. Hildesheimer Informatikberichte 1/89, Universität Hildesheim, Institut für Informatik, 1989.
8. J. Ezpeleta, J.M. Couvreur, and M. Silva. A new technique for finding a generating family of siphons, traps and st-components. application to colored petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, LNCS 674, pages 126–147. Springer, Berlin, 1993.
9. G.H. Golub and C.F. Van Loan. *Matrix Computations*. The John Hopkins Press, Baltimore, 2nd edition, 1989.
10. P. Kemper. Linear time algorithm to find a minimal deadlock in a strongly connected free-choice net. In M. Ajmone-Marsan, editor, *Application and Theory of Petri Nets 1993*, LNCS 691, pages 319–338, Berlin, 1993. Springer.
11. 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.
12. R. Tarjan. Depth-first search and linear graph algorithms. In *SIAM. J. Comp.*, volume 1, pages 146–160. 1972.

This article was processed using the LaTeX macro package with LLNCS style