Fundamenta Informaticae XX (2015) 1–21
DOI 10.3233/FI-2012-0000
IOS Press

Constructing Minimal Coverability Sets

Artturi Piipponen, Antti Valmari
Department of Mathematics, Tampere University of Technology
P.O. Box 553, FI–33101 Tampere, FINLAND
artturi.piipponen@tut.fi, antti.valmari@tut.fi

Abstract. This publication addresses two bottlenecks in the construction of minimal coverability sets of Petri nets: the detection of situations where the marking of a place can be converted to ω, and the manipulation of the set A of maximal ω-markings that have been found so far. For the former, a technique is presented that consumes very little time in addition to what maintaining A consumes. It is based on Tarjan's algorithm for detecting maximal strongly connected components of a directed graph. For the latter, a data structure is introduced that resembles BDDs and Covering Sharing Trees, but has additional heuristics designed for the present use. Results from a few experiments are shown. They demonstrate significant savings in running time and varying savings in memory consumption compared to an earlier state-of-the-art technique.

Keywords: coverability set, Tarjan's algorithm, antichain data structure

1. Introduction

A *place/transition net* is a Petri net whose places may contain any finite number of tokens, and tokens are indistinguishable from each other. When people talk about Petri nets without specifying the precise net class, they often mean place/transition nets. The "state" or "configuration" of a place/transition net is called *marking*. It tells the number of tokens in each place. Markings are changed by *firing* transitions. A marking M is *reachable* if and only if some finite sequence of firings of transitions transforms the initial marking of the net to M. All these notions will be made precise in Section 2.

A place/transition net is finite if and only if its sets of places and transitions are finite. The set of reachable markings of a finite place/transition net is not necessarily finite. This is because with some nets, the number of tokens in one or more places grows without limit. However, Karp and Miller proved that if this growth is approximated from above using a certain kind of extended markings, then a finite set may always be obtained [8]. This set is called *coverability set*. The extended markings may use the symbol ω instead of a natural number to denote the number of tokens in a place. They do not have an established name, but it is natural to call them *ω-markings*. Intuitively, ω denotes "unbounded". An ω-marking M *covers* an ω-marking M' if and only if, for every place p, M has at least the same number of tokens in p as M' has, where ω is considered bigger than any natural number. For every reachable marking M, a coverability set contains an ω-marking that covers M. Every ω-marking in the set is either a reachable marking or a limit of a growing infinite sequence of reachable markings.

The coverability set, as defined by Karp and Miller, is not necessarily unique. Finkel proved that there always is a unique *minimal coverability set* [6]. It is always finite and has no other coverability set as a subset. It is obtained from a coverability set by removing each ω-marking that is not *maximal*. An ω-marking is maximal if and only if it is not strictly covered by another ω-marking in the set.

A basic step in the construction of coverability sets consists of firing a transition at an ω-marking and then changing the markings of zero or more places from a finite value to ω. The details of adding the ω-symbols vary between different coverability set construction algorithms. The construction starts at the initial ω-marking and proceeds by constructing new ω-markings with the basic step. New ω-markings that are covered by already found ω-markings are rejected sooner or later, depending on the algorithm. Eventually new non-covered ω-markings are no longer found, and the construction terminates.

The construction of coverability sets is computationally challenging. First, the result may be huge. This problem may be alleviated by constructing the minimal coverability set, but not eliminated altogether. Also the minimal coverability set may be huge. Even when it is not, an algorithm that constructs it may construct a huge number of ω-markings that are later removed because of becoming strictly covered by more recently constructed ω-markings.

Second, all known minimal coverability set algorithms maintain in one form or another a set A of maximal ω-markings found so far. Here "A" stands for "antichain". Nobody knows how to implement this set such that all its operations are always fast.

Third, all known coverability set algorithms check a so-called *pumping condition* every now and then. The *finding predecessor* of an ω-marking M is the ω-marking from which M was constructed for the first time. The initial ω-marking does not have a finding predecessor. The *finding history* of M consists of its finding predecessor, the finding predecessor of the finding predecessor, and so on up to the initial ω-marking. The *full history* is defined otherwise similarly, but every, not just the first, instance of constructing the ω-marking is taken into account. The basic form of the pumping condition consists of checking whether the most recently found ω-marking M covers any ω-marking in the finding history of M. Alternatively, the full history can be used. Intuitively, with the full history, valid pumping conditions would be found more often and thus termination would be reached with fewer firings of transitions. Checking whether M covers M' is easy, but the finding history and especially the full history may be big.

Somewhat complicated ideas for speeding up the construction of minimal coverability sets have been suggested [6, 7, 10]. They mostly aim at reducing the number of ω-markings that are constructed but rejected later. They address the organization of the work at a higher level than the implementation of the pumping condition and the set A. A main idea in them is known as "pruning".

The publications [12, 13] gave both theoretical, heuristic, and experimental evidence that at the higher level, a straightforward approach is very competitive, if ω-markings are constructed in depth-first or so-called most tokens first order. They demonstrate that the pruning sometimes works against its purpose, weakening the performance. They prove two theorems saying that if the full histories are used for detecting pumping conditions, then the benefit that the pruning aims at comes automatically, without explicitly implementing the pruning and thus without suffering from the above-mentioned weakness.

The theorems in [12, 13] do not prove that the straightforward approach with full histories is always at least as good as pruning. This is because it was observed in [12, 13] that the running time of their algorithm sometimes depends dramatically on the order in which the transitions are listed in the input. The same seems likely also for other algorithms. That is, a detail that has nothing to do with the algorithms themselves may dominate performance measurements. Sorting the transitions according to a natural heuristic did not solve the problem, because the sorted order was good with some and bad with some other Petri nets. So in this field, one must not trust too much on measurements. This makes it difficult to compare the relative merits of the algorithms. Even so, the available evidence suggests strongly that the algorithm in [12, 13] significantly outperforms the earlier algorithms.

At a low level, the most time-consuming operations in the algorithm in [12, 13] are the two operations mentioned above, that is, the manipulation of A and the detection of pumping conditions. This publication presents a significant improvement to both. Again, the obtained reduction in running time is so big that, despite the problem with measurements mentioned above, it seems safe to conclude that the algorithm in this publication tends to be significantly faster than all earlier algorithms. Memory-wise the new algorithm is roughly as good as the one in [12, 13].

In a sense, this publication reverses the order in which the pumping condition is checked. In most if not all earlier publications, the finding history, the full history, or some other subset of the ω-markings M' that have a path to the current ω-marking M are found, and it is tested whether M covers any of them. In this publication, the ω-markings M' in A that are strictly covered by M are found, and it is tested whether any of them has a path to M. In both approaches, to maintain A, it seems necessary to find the strictly covered ω-markings M'. The new algorithm exploits this work that has to be done anyway. It implements the pumping condition with only a constant amount of extra work per each strictly covered M'. This should be contrasted to the earlier approaches, where the finding of the relevant ω-markings is much more additional work on top of the updating of A.

The new algorithm detects the pumping condition for the intersection of the full history with A. Fortunately, Theorem 5.3 in this publication tells that this yields precisely the same ω-markings as the full history would. Therefore, the two theorems in [12, 13] stay valid.

Section 2 defines the necessary concepts. The overall structure of our new algorithm is presented in Section 3. In Section 4 a sufficient condition is discussed that guarantees that the algorithm has already covered (instead of just will cover) the "future" of an ω-marking. Section 5 describes how *Tarjan's algorithm* for detecting maximal strongly connected components of a directed graph [11, 1, 5] can be harnessed to check in constant time whether there is a path from M' to M. A data structure that improves the efficiency of maintaining A is introduced in Section 6. It uses ideas from BDDs [3] and covering sharing trees [4], and has heuristics designed for minimal coverability sets. An additional optimisation is discussed in Section 7. Section 8 presents a few performance measurements without and with using the new ideas.

This publication is an extended version of [9]. The main differences are that more measurements are reported and significant effort has been put to clarify the proofs of Corollary 4.6 and Theorem 5.3. A difficulty with both intuitive understanding and formal proofs of coverability set algorithms is that, unlike with Petri nets, if M_2 is obtained from M_1 by firing transition t, similarly with M_2' and M_1', and $M_1' > M_1$, then we cannot conclude that $M_2' > M_2$. Much worse, we cannot even conclude that $M_2' \geq M_2$. This is because the construction of M_2 may involve addition of ω-symbols that are not added when constructing M_2' (and that were not in M_1' to start with). Corollary 4.6 presents a sufficient condition that guarantees $M_2' \geq M_2$. Originally, Corollary 4.6 was used as a lemma in the proof of Theorem 5.3. The new proof of the theorem does not need it. However, because Corollary 4.6 presents an intuitively interesting fact that may be useful when developing new methods for answering verification questions during the construction of the set, we chose to keep it in the present publication. An example of its use is given in Section 4.

2. Concepts and Notation

The set of natural numbers is $\mathbb{N} = \{0, 1, 2, \ldots\}$. If X is a set, then the set of finite sequences of elements of X is denoted with X^*. The empty sequence is denoted with ε.

A finite *place/transition net* is a tuple $(P, T, W, \hat{M})$, where P and T are finite sets, $P \cap T = \emptyset$, W is a function from $(P \times T) \cup (T \times P)$ to $\mathbb{N}$, and $\hat{M}$ is a function from P to $\mathbb{N}$. The elements of P, T, and W are called *places*, *transitions*, and *weights*, respectively. A *marking* M is a function from P to $\mathbb{N}$. It can also be thought of as a vector of $|P|$ natural numbers. Thus $\hat{M}$ is a marking. It is called the *initial marking*.

A transition t is *enabled at* M, denoted with $M\,[t\rangle$, if and only if $M(p) \geq W(p,t)$ for every $p \in P$. Then t may *occur* yielding the marking M' such that $M'(p) = M(p) - W(p,t) + W(t,p)$ for every $p \in P$. This is denoted with $M\,[t\rangle\,M'$. It is also said that t is *fired* at M yielding M'. The notation is extended to finite sequences of transitions in the natural way: $M\,[t_1t_2\cdots t_n\rangle\,M'$ if and only if there are $M_0, M_1, \ldots, M_n$ such that $M = M_0$, $M_{i-1}\,[t_i\rangle\,M_i$ for $1 \leq i \leq n$, and $M_n = M'$.

The notation $\Delta_{t_1\cdots t_n}(p)$ is defined by $\Delta_{t_1\cdots t_n}(p) = \sum_{i=1}^{n} W(t_i, p) - W(p, t_i)$. Clearly $\Delta_{t_1\cdots t_n}(p)$ is finite. The equation $M' = M + \Delta_{t_1\cdots t_n}$ denotes the claim that for every $p \in P$ we have $M'(p) = M(p) + \Delta_{t_1\cdots t_n}(p)$. If $M\,[t_1t_2\cdots t_n\rangle\,M'$, then $M' = M + \Delta_{t_1\cdots t_n}$ holds.

A marking M' is *reachable from* M if and only if there is $\sigma \in T^*$ such that $M\,[\sigma\rangle\,M'$. The set of *reachable markings* is $\{M \mid \exists \sigma \in T^* : \hat{M}\,[\sigma\rangle\,M\}$, that is, the set of markings that are reachable from the initial marking.

An ω*-marking* is a vector of $|P|$ elements of the set $\mathbb{N} \cup \{\omega\}$. The enabledness and occurrence rules of transitions are extended to ω-markings with the conventions that for every $i \in \mathbb{N}$, $\omega > i$ and $\omega + i = \omega - i = \omega$. An ω-marking is a marking if and only if it contains no ω-symbols. From now on, M, M', M_1, and so on are ω-markings, unless otherwise stated. If $M\,[t_1t_2\cdots t_n\rangle\,M'$, then $M'(p) = M(p) + \Delta_{t_1\cdots t_n}(p)$ holds even if $M(p) = \omega$, and then $M'(p) = \omega$.

We say that M' *covers* M and write $M \leq M'$ if and only if $M(p) \leq M'(p)$ for every $p \in P$. We say that M' *strictly covers* M and write $M < M'$ if and only if $M \leq M'$ and $M \neq M'$. We say that M is *a limit* of a set $\mathcal{M}$ of ω-markings if and only if for every $i \in \mathbb{N}$, $\mathcal{M}$ contains an M_i such that $M_0 \leq M_1 \leq \ldots$ and for every $p \in P$, either $M(p) = \omega$ and $M_i(p)$ grows without limit as i grows, or there is i such that $M(p) = M_i(p) = M_{i+1}(p) = \ldots$. This definition may be a bit unexpected, because it allows the use of the same ω-marking many times in $M_0 \leq M_1 \leq \ldots$. (So our limits are not precisely the same thing as "limit points" in topology.) As a consequence, every element of $\mathcal{M}$ is a limit of $\mathcal{M}$, because obviously $M \leq M \leq \ldots$. On the other hand, a set may also have limits that are not in the set.

A *coverability set* is any set $\mathcal{M}$ that satisfies the following conditions:

1. Every reachable marking M is covered by some $M' \in \mathcal{M}$.
2. Every $M \in \mathcal{M}$ is a limit of reachable markings.

A coverability set is not necessarily finite. Indeed, the set of reachable markings is always a coverability set, and it may be infinite. Fortunately, there is a unique *minimal coverability set* that is always finite [6]. It has no other coverability set as a subset, and no ω-marking in it is covered by another ω-marking in it. It consists of the maximal elements of the set of the limits of the set of reachable markings. Proofs of these facts are not trivial. Among others, [12, 13] contain an attempt to make the proofs accessible.

The construction of coverability sets resembles the construction of the set of reachable markings but has additional features. The basic step of the construction starts with an already found ω-marking M. A transition t is fired at M, yielding M'. A central idea is that if there are M_0 and σ such that $M_0 < M'$ and $M_0 \left[\sigma\right\rangle M$, then the sequence σt can occur repeatedly without limit, making the markings of those p that have $M'(p) > M_0(p)$ grow without limit, while the remaining p have $M'(p) = M_0(p)$. The limit of the resulting markings is M'', where $M''(p) = \omega$ if $M'(p) > M_0(p)$ and $M''(p) = M_0(p)$ otherwise.

Roughly speaking, instead of storing M' and remembering that $M \left[t\right\rangle M'$, most if not all algorithms store M'' and remember that $M \mathrel{-t\overset{\omega}{\rightarrow}} M''$, where $M \mathrel{-t\overset{\omega}{\rightarrow}} M''$ denotes that M'' was obtained by firing t at M and then possibly adding ω-symbols to the result as was discussed above. However, although this rough description of the basic step of the algorithms serves well as a starting point, it is too imprecise for deeper discussion. There are four issues.

First, the algorithms need not remember that $M \mathrel{-t\overset{\omega}{\rightarrow}} M''$. It suffices to remember that there is t such that $M \mathrel{-t\overset{\omega}{\rightarrow}} M''$.

Second, instead of $M_0 \left[\sigma\right\rangle M$ in the above, the algorithms use $M_0 \mathrel{-\sigma\overset{\omega}{\rightarrow}} M$, because they only have access to the latter. Here $-\sigma\overset{\omega}{\rightarrow}$ is the natural extension of $-t\overset{\omega}{\rightarrow}$: $M \mathrel{-t_1 t_2 \cdots t_n\overset{\omega}{\rightarrow}} M'$ if and only if there are $M_0, M_1, \ldots, M_n$ such that $M = M_0$, $M_{i-1} \mathrel{-t_i\overset{\omega}{\rightarrow}} M_i$ for $1 \leq i \leq n$, and $M_n = M'$.

Third, after firing t at M, an algorithm may use more than one M_0 and σ that have $M_0 \mathrel{-\sigma\overset{\omega}{\rightarrow}} M$ to add ω-symbols. For instance, if $(2,0) \left[t_1\right\rangle (0,1) \left[t\right\rangle (1,1)$, then $(0,1)$ and ε justify converting $(1,1)$ to $(\omega,1)$, after which $(2,0)$ and t_1 justify further conversion to (ω,ω). To discuss this, let us first introduce some terminology.

Assume that M' is obtained by firing t at M and then performing zero or more "pumping operations" that are defined soon. We say that the *pumping condition semi-holds* with M_0 if and only if $M_0 < M'$ and there is $\sigma \in T^*$ such that $M_0 \mathrel{-\sigma\overset{\omega}{\rightarrow}} M$. The pumping condition *holds* if and only if it semi-holds and there is $p \in P$ such that $M_0(p) < M'(p) < \omega$. The *pumping operation* consists of assigning ω to those $M'(p)$ that have $M_0(p) < M'(p) < \omega$.

The algorithms perform the pumping operation only when the pumping condition holds. After it, the pumping condition semi-holds but does not hold. Some algorithms in [12, 13] and all new algorithms in this publication never fail to perform the pumping operation when the pumping condition holds, but this is not necessarily true of all algorithms.

As a consequence of the pumping operation, the pumping condition may start to hold with another M_0 with which it did not originally hold. This may trigger a new pumping operation.

Fourth, after firing $M \left[t\right\rangle M'$ and doing zero or more pumping operations, an algorithm may reject the resulting M', if it is covered by some already stored ω-marking M''. The intuition is that whatever M' could contribute to the minimal coverability set, is also contributed by M''. So M' need not be investigated.

Whether $M \mathrel{-t\overset{\omega}{\rightarrow}} M'$ holds depends on not just M, t, and M', but also on what the algorithm has done before trying t at M. So the precise meaning of the notation $M \mathrel{-t\overset{\omega}{\rightarrow}} M'$ depends on the particular algorithm and may even depend on the order in which the places and transitions are listed in the input to the algorithm. We emphasize that this phenomenon is not specific to our algorithm but affects coverability set construction algorithms in general. Most publications leave this issue implicit, trusting that the readers get sufficient information from the description of the algorithm. The precise meaning of $M \overset{\omega}{-t\rightarrow} M'$ used in this publication will be made explicit in Definition 3.1. The definition is technical, but intuitively it is the same concept as in essentially all publications on the construction of coverability sets: M' is what the algorithm produces by first firing t at M and then possibly adding ω-symbols.

3. Overall Algorithm

Figure 1 shows the new minimal coverability set construction algorithm of this publication in its basic form. Variants of it will be discussed in Sections 7 and 8.

Lines 1, 3–6, 13–16, 18, 20–24, and 27 implement most of the minimal coverability set construction algorithm of [12, 13], specialized to depth-first search. Let us discuss them in this section. The remaining lines are grey to indicate that they may be ignored until they are discussed in later sections. (The pseudocode in [12, 13] used a generic search discipline. Breadth-first search, depth-first search, and so-called most tokens first search were studied in detail. Some details related to the implementation of the pumping operation have been replaced in this publication by another mechanism.)

The set A contains the maximal ω-markings that have been found so far. Upon termination it contains the result of the algorithm. Its implementation will be discussed in Section 6. In addition to what is explicit in Figure 1, the call on line 22 may remove elements from A in favour of a new element M' that strictly covers them. We will discuss this in detail later.

The set F is a hash table. ω-markings are added to it at the same time as to A, but they are never removed from it. So always $A \subseteq F$. The attributes of an ω-marking such as $M.\mathsf{tr}$ (discussed soon) are stored in F and not in A. That is, F contains records, each of which contains an ω-marking and some additional information. One reason is that, as we will see later, some information on an ω-marking remains necessary even after the ω-marking has been removed from A. Another reason is that, as will become obvious in Section 6, the data structure that is discussed there cannot associate attributes to the ω-markings that it stores. Like in [12, 13], F is also used to implement an optimisation that will be discussed together with lines 18 and 20. Hash tables are very efficient, so F does not cause significant extra cost.

For efficiency, instead of the common recursive implementation, depth-first search is implemented with the aid of a stack which is called W (for work-set). (So the symbol W is used both for the weight function of Petri nets and for the workset of the algorithm, but these uses are so remote from each other that confusion seems unlikely.) The elements of W are pointers to records in F. Each ω-marking M has an attribute tr that points to the next transition that should be tried at M.

The algorithm starts on lines 1 and 2 by putting the initial marking to all data structures. Roughly speaking, lines 3 to 27 try each transition t at each encountered ω-marking M in depth-first order. (This is not strictly true, because heuristics that are discussed later may prematurely terminate the processing of M and may cause the skipping of some transitions at M.) If M has untried transitions, line 4 picks the next, otherwise lines 6–13 that implement backtracking are executed. Lines 7–12 will be discussed later.

If the picked transition t is disabled at the current ω-marking M, then it is rejected on line 15. Otherwise t is fired at M on line 16. Lines 17 and 19 will be discussed later. If M' has already been encountered, it is rejected on lines 18 and 20. This quick rejection of M' is useful, because reaching the

```
 1  $F := \{\hat{M}\}$; $A := \{\hat{M}\}$; $W$.push($\hat{M}$); $\hat{M}$.tr := first transition
 2  $S$.push($\hat{M}$); $\hat{M}$.ready := false; $n_f := 1$; $\hat{M}$.index := 1; $\hat{M}$.lowlink := 1
 3  while $W \neq \emptyset$ do
 4      $M := W$.top; $t := M$.tr; if $t \neq$ nil then $M$.tr := next transition endif
 5      if $t =$ nil then
 6          $W$.pop
 7          activate transitions as discussed in Section 7
 8          if $M$.lowlink $= M$.index then
 9              while $S$.top $\neq W$.top do $S$.top.ready := true; $S$.pop endwhile
10          else
11              $W$.top.lowlink := min{$W$.top.lowlink, $M$.lowlink}
12          endif
13          go to line 3
14      endif
15      if $\neg M\,[t\rangle$ then go to line 3 endif
16      $M'$ := the $\omega$-marking such that $M\,[t\rangle\,M'$
17      if $M' \leq M$ then passivate $t$; go to line 3 endif
18      if $M' \in F$ then
19          if $\neg M'$.ready then $M$.lowlink := min{$M$.lowlink, $M'$.lowlink} endif
20          go to line 3
21      endif
22      Cover-check($M'$, $A$)      // only keep maximal — may update $A$ and $M'$
23      if $M'$ is covered by an element of $A$ then go to line 3 endif
24      $F := F \cup \{M'\}$; $A := A \cup \{M'\}$; $W$.push($M'$); $M'$.tr := first transition
25      $S$.push($M'$); $M'$.ready := false
26      $n_f := n_f + 1$; $M'$.index := $n_f$; $M'$.lowlink := $n_f$
27  endwhile
```

Figure 1. A minimal coverability set algorithm that uses Tarjan's algorithm and some heuristics

same ω-marking again is expected to be very common, because $M\,[t_1t_2\rangle\,M_{12}$ and $M\,[t_2t_1\rangle\,M_{21}$ imply that $M_{12} = M_{21}$. Without lines 18 and 20, M' would be rejected on line 23, but after consuming more time. Line 18 is also needed because of line 19.

The call Cover-check(M', A) first checks whether M' is covered by some ω-marking in A. The details of this are discussed in Section 6. If M' is covered, then M' is rejected on line 23.

In the opposite case, Cover-check checks whether the pumping condition holds with any $M_0 \in A$. (In [12, 13], the pumping condition was detected for $M_0 \in F$. Theorem 5.3 will tell why it suffices to use A instead.) If yes, it changes $M'(p)$ to ω for the appropriate places p. Cover-check also removes from A those ω-markings that the updated M' covers strictly. When M is removed, M.tr is set to nil, so that even if the algorithm backtracks to M in the future, no more transitions will be fired at it. The addition of ω-symbols makes M' grow in the "$\leq$" ordering and may thus make the pumping condition hold with some other M_0. Cover-check continues until there is no $M_0 \in A$ with which the pumping condition holds. How Cover-check finds those $M_0 \in A$ that have $M_0 < M'$ is discussed in Section 6.

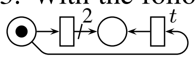
The checking of the existence of a path from M_0 to M is explained in Section 5.

If M' was not covered, its updated version is added to the data structures on lines 24–26. This implements the entering to M' in the depth-first search.

It is the time to make the notation $M \,{-}t\overset{\omega}{\rightarrow}\, M'$ precise.

Definition 3.1. $M \,{-}t\overset{\omega}{\rightarrow}\, M'$ denotes that t was fired at M on line 16 resulting in some M'' such that $M'' \not\leq M$, and either $M'' \in F$ held on line 18 (in which case $M' = M''$), or M'' was transformed to M' on line 22 and then added to F on line 24.

Thus $M \,{-}t\overset{\omega}{\rightarrow}\, M'$ is either always false, or becomes true during the execution of the algorithm and never becomes false again. When it holds, then $M \in F$ and $M' \in F$. The algorithm will consider the edge $M \,{-}t\overset{\omega}{\rightarrow}\, M'$ when detecting pumping conditions, as will be discussed in Section 5. If M' is rejected on line 17 or 23, then $M \,{-}t\overset{\omega}{\rightarrow}\, M'$ does not become true, $M' \in F$ may fail to hold, and the algorithm forgets the result of firing t at M. The correctness of this forgetting on line 17 will be discussed in Section 7 (and is intuitively obvious). The case of line 23 is much trickier. It follows from the correctness of the algorithm as a whole, which was proven in [12, 13].

Even if $M \in F$ and $M\,[t\rangle$, it may be that there never is any M' such that $M \,{-}t\overset{\omega}{\rightarrow}\, M'$. This is the case if M is removed from A and $M.\mathsf{tr}$ is set to nil before t is tried at M, or if the result of trying t at M is rejected on line 17 or 23. With the following Petri net, the latter happens although $M \in A$ when the algorithm has terminated: 

.

Ignoring the lines added in this publication (and without restricting to depth-first search discipline), the correctness of the algorithm has been proven in detail in [12, 13]. One detail of the proof will be used in later sections of the present publication, so we present it as a lemma. It holds always during and after the execution of the algorithm, except in the middle of the updating operations mentioned in its proof.

Lemma 3.2. For every $M_0 \in F$, there is $M_0' \in A$ such that $M_0 \leq M_0'$.

Proof:
Each time when an ω-marking M_0 is inserted to F, it is also inserted to A. Each time when an M_0 is removed from A, an M_0' such that $M_0 < M_0'$ is inserted to A. Therefore, the algorithm maintains the claimed property as an invariant. □

4. Ripe ω-markings

Lemma 3.2 guarantees that all ω-markings that can be reached by firing transition sequences starting at a found ω-marking M, *will eventually be covered* by the ω-markings found by the algorithm. If $M \notin A$ and the firing of some transitions has not yet been tried at M, the lemma justifies leaving them untried forever. The goal of this section is to present and prove correct a condition that guarantees that all ω-markings that can be reached by firing transition sequences starting at M, *have already now been covered* by the ω-markings found by the algorithm.

To liberate the symbol M for other uses, from now on we denote the M of the algorithm with M_c and call it the *current* ω-marking. The same ω-marking may be current several times, because the search may continue to other ω-markings and later backtrack to it.

The following lemma follows trivially from the transition firing rule and the fact that after firing a transition, the marking of a place may be converted from a finite value to ω but not vice versa.

Lemma 4.1. If $M \,{-}\sigma\overset{\omega}{\rightarrow}\, M'$ and $M(p) = \omega$, then $M'(p) = \omega$.

In depth-first search, an ω-marking M is *white* if it has not been found (that is, $M \notin F$); *grey* if it has been found but not backtracked from (that is, $M \in W$); and *black* if it has been backtracked from. The grey ω-markings M_i^{g} and the $M_{i-1}^{\mathsf{g}} \,{-}t_i\overset{\omega}{\rightarrow}\, M_i^{\mathsf{g}}$ via which they were first found constitute a path from the initial ω-marking $M_0^{\mathsf{g}} = \hat{M}$ to M_{c}.

Definition 4.2. An ω-marking M is *fresh-ripe* if and only if $M \in A$, and either the algorithm has terminated, or currently for some place p_{r} we have $M_{\mathsf{c}}(p_{\mathsf{r}}) < \omega = M(p_{\mathsf{r}})$. It is *ripe* if and only if it is or has been fresh-ripe.

A fresh-ripe ω-marking may cease from being fresh-ripe by being removed from A or by the current ω-marking changing such that $M_{\mathsf{c}}(p_{\mathsf{r}}) < \omega$ no longer holds. A ripe ω-marking obviously remains ripe until the termination of the algorithm. The following lemmas tell more properties of (fresh-)ripe ω-markings.

Lemma 4.3. If M is ripe, then the algorithm has tried every transition at M.

Proof:
It suffices to prove the claim for an arbitrary fresh-ripe M, because the promised property obviously remains valid afterwards. It is obvious from Figure 1 that after an ω-marking has been removed from A, it is never again put to A. Because $M \in A$, the investigation of transitions at M has not been prematurely terminated by setting $M.\mathsf{tr}$ to nil. Therefore, if the algorithm has not tried every transition at M, then M is grey. So the algorithm has not terminated and there is a path from M to M_{c}. Lemma 4.1 yields a contradiction with $M_{\mathsf{c}}(p_{\mathsf{r}}) < \omega = M(p_{\mathsf{r}})$. □

Lemma 4.4. If M_0' is fresh-ripe, $M_0' \geq M_0$, and $M_0\,[t_1\rangle\, M_1\,[t_2\rangle \cdots [t_n\rangle\, M_n$, then for $1 \leq i \leq n$ there are fresh-ripe M_i' such that $M_i \leq M_i'$ and $M_i' \geq M_{i-1}' + \Delta_{t_i} \geq M_0' + \Delta_{t_1 \cdots t_i}$.

Proof:
We use induction on $0 \leq i \leq n$. The base case $i = 0$ only says that $M_0' \geq M_0$. It obviously holds.

Assume that the claim holds with $i-1$. By Lemma 4.3, the algorithm has tried t_i at M_{i-1}'. That yielded a result M_i''' such that $M_i''' \geq M_i$, because $M_{i-1}' \geq M_{i-1}$ and $M_{i-1}\,[t_i\rangle\, M_i$. For any $p \in P$, if $M_i'''(p)$ was converted to ω, then $M_i'''(p) = \omega \geq M_{i-1}'(p) + \Delta_{t_i}(p)$, otherwise $M_i'''(p) = M_{i-1}'(p) + \Delta_{t_i}(p)$. If M_i''' was added to A, we choose $M_i'' = M_i'''$, otherwise M_i''' was rejected because it was covered by an ω-marking in A which we call M_i''. In both cases $M_i''' \leq M_i''$. Since then, M_i'' may have been removed from A, but by Lemma 3.2, there is $M_i' \in A$ such that $M_i \leq M_i''' \leq M_i'' \leq M_i'$.

We have $M_i' \geq M_i''' \geq M_{i-1}' + \Delta_{t_i} \geq M_0' + \Delta_{t_1 \cdots t_{i-1}} + \Delta_{t_i} = M_0' + \Delta_{t_1 \cdots t_i}$. If the algorithm has terminated, then M_i' is fresh-ripe. Otherwise there is $p_{\mathsf{r}} \in P$ such that $M_{\mathsf{c}}(p_{\mathsf{r}}) < \omega = M_0'(p_{\mathsf{r}})$. By the above inequality $M_i'(p_{\mathsf{r}}) \geq \omega + \Delta_{t_1 \cdots t_i}(p_{\mathsf{r}})$, implying $M_i'(p_{\mathsf{r}}) = \omega$. So M_i' is fresh-ripe. In conclusion, M_i' has the required properties. □

Lemma 4.5. If M_0' is fresh-ripe, $M_0' \geq M_0$, and $M_0 \,{-}t_1\overset{\omega}{\rightarrow}\, M_1 \,{-}t_2\overset{\omega}{\rightarrow} \cdots {-}t_n\overset{\omega}{\rightarrow}\, M_n$, then for $1 \leq i \leq n$ there are fresh-ripe M_i' such that $M_i \leq M_i'$ and $M_i' \geq M_{i-1}' + \Delta_{t_i} \geq M_0' + \Delta_{t_1 \cdots t_i}$.

Proof:
Like in the proof of Lemma 4.4, we use induction on $0 \leq i \leq n$. The base case $i = 0$ obviously holds.

Assume that M'_{i-1} is fresh-ripe and $M_{i-1} \leq M'_{i-1}$. We have to prove that there is a fresh-ripe M'_i such that $M_i \leq M'_i$ and $M'_i \geq M'_{i-1} + \Delta_{t_i}$. The situation is different from Lemma 4.4, because with $M_{i-1} -t_i\overset{\omega}{\rightarrow} M_i$, there may be $p_1, \ldots, p_k$ such that $M_{i-1}(p_j) < \omega = M_i(p_j)$. Therefore, we apply induction on $0 \leq j \leq k$. We may assume that $p_1, \ldots, p_k$ are listed in the order in which the algorithm made $M_i(p_j) = \omega$ hold. Let $M_{i,j}$ be the ω-marking just after the algorithm made $M_i(p_j) = \omega$. So $M_{i-1} [t_i\rangle\, M_{i,0}$ and $M_{i,k} = M_i$. For $0 \leq j \leq k$, we will show the existence of a fresh-ripe $M'_{i,j}$ such that $M_{i,j} \leq M'_{i,j}$. Actually, these $M'_{i,j}$ will prove equal, but the subscript j helps to follow the proof.

Because M'_{i-1} is fresh-ripe, $M_{i-1} \leq M'_{i-1}$, and $M_{i-1} [t_i\rangle\, M_{i,0}$, Lemma 4.4 yields a fresh-ripe $M'_{i,0}$ such that $M_{i,0} \leq M'_{i,0}$ and $M'_{i,0} \geq M'_{i-1} + \Delta_{t_i}$. This is the base case $j = 0$.

For the induction step on j, assume that $M_{i,j-1} \leq M'_{i,j-1}$ and $M'_{i,j-1}$ is fresh-ripe. Let σ_j be the pumping sequence that justified converting $M_{i,j-1}$ to $M_{i,j}$. For any $p \in P$, if $M_{i,j-1}(p) < \omega$, then $\Delta_{\sigma_j}(p) \geq 0$. Let σ_j^h be σ_j repeated h times, where $h \in \mathbb{N}$. There is $M_{i,j,h}$ such that $M_{i,j-1} [\sigma_j^h\rangle\, M_{i,j,h} \geq M_{i,j-1}$. Lemma 4.4 yields a fresh-ripe $M'_{i,j,h}$ such that $M_{i,j,h} \leq M'_{i,j,h}$ and $M'_{i,j,h} \geq M'_{i,j-1} + \Delta_{\sigma_j^h}$. For any $p \in P$, if $M_{i,j-1}(p) < \omega$, then $M'_{i,j,h}(p) \geq M'_{i,j-1}(p)$, and otherwise $M'_{i,j-1}(p) = \omega = M'_{i,j,h}(p)$. So $M'_{i,j,h} \geq M'_{i,j-1}$.

Because $M'_{i,j-1}$ and $M'_{i,j,h}$ are fresh-ripe, they all belong to A. Because A is the set of maximal ω-markings found so far, we have $M'_{i,j-1} \not< M'_{i,j,h}$. So $M'_{i,j-1} = M'_{i,j,h}$. This holds for every $h \in \mathbb{N}$. If $M_{i,j}(p) = \omega > M_{i,j-1}(p)$, then $M_{i,j,h}(p)$ grows without limit as h grows, implying $M'_{i,j,h}(p) = \omega$. With the remaining p, $M'_{i,j,h}(p) = M'_{i,j-1}(p) \geq M_{i,j-1}(p) = M_{i,j}(p)$. These yield $M_{i,j} \leq M'_{i,j,h}$. So $M'_{i,j,h} = M'_{i,j-1}$ qualifies as the $M'_{i,j}$. The induction proof on j is ready.

Choosing $M'_i = M'_{i,k}$ completes the proof of induction step i. Because $M'_{i,k} = M'_{i,k-1} = \ldots = M'_{i,0}$, we have $M'_i \geq M'_{i-1} + \Delta_{t_i}$. □

A fresh-ripe ω-marking may cease from being fresh-ripe. This restricts the applicability of Lemmas 4.4 and 4.5. To alleviate this problem, the main message of the lemmas is formulated below anew in such a form that after the ω-marking has become ripe, the claim applies continuously up to the termination of the algorithm. The $[t_i\rangle$ -claim tells that whatever *the net could do* from M_0 has been covered by the algorithm, and the $-t_i\overset{\omega}{\rightarrow}$-claim tells that whatever *the algorithm has done* from M_0 has been covered by the algorithm. The latter is not necessarily a subset of the former, because the net cannot but the algorithm can change the marking of a place from a finite value to ω. In both cases, the algorithm has completed the investigation of the part of the coverability set that is reachable from M_0 or M'_0.

Corollary 4.6. If M'_0 is ripe, $M'_0 \geq M_0$, and $M_0 [t_1\rangle\, M_1 [t_2\rangle \cdots [t_n\rangle\, M_n$ or $M_0 -t_1\overset{\omega}{\rightarrow} \cdots -t_n\overset{\omega}{\rightarrow} M_n$, then for $1 \leq i \leq n$ there are ripe $M'_i \in F$ such that $M_i \leq M'_i$ and $M'_i \geq M'_{i-1} + \Delta_{t_i} \geq M'_0 + \Delta_{t_1 \cdots t_i}$. The algorithm will never again try transitions at any M''_i that has $M''_i \leq M'_i$.

Proof:
Lemmas 4.4 and 4.5 promise the existence of M'_i at the time when M'_0 becomes fresh-ripe, and the promised properties of M'_i remain valid from then on. It is an elementary property of the algorithm that it does not try any transitions at ω-markings that are not in A, and no M''_i such that $M''_i < M'_i$ can enter A. By Lemma 4.3, the algorithm will not try any transitions at M'_i while it stays in A. □

As an example of an application of the corollary, let transitions enter and leave model a pedestrian entering and leaving a zebra crossing, and let truck model a truck driving through the zebra crossing. We want to verify that after each occurrence of enter, truck does not occur before leave has occurred. As demonstrated with an example in Section 3, the algorithm does not necessarily keep track of all firings of transitions. So the verification should be based on firing transitions anew at found ω-markings. To find errors as soon as possible, we do not want to first wait until the termination of the construction of the minimal coverability set. Instead, we want to do the check every now and then during the construction. Definition 4.2 and Corollary 4.6 tell when it is reasonable to do such a check.

5. Constant-time Reachability Testing

A *maximal strongly connected component* or *strong component* of a directed graph (V, E) is a maximal set of vertices $V' \subseteq V$ such that for any two vertices u and v in V', there is a path from u to v. The strong components constitute a partition of V. Tarjan's algorithm [11, 1] detects strong components in time $O(|V| + |E|)$. It is based on depth-first search of the graph. It is slower than depth-first search only by a small constant factor. A particularly good description of an optimised version of it is in [5].

In our case, V consists of all ω-markings that are encountered and not rejected during the construction of the minimal coverability set, that is, those that are (eventually) stored in F. The edges are defined by $(M, M') \in E$ if and only if there is $t \in T$ such that (eventually) $M -t\overset{\omega}{\rightarrow} M'$. This notion, and thus also V and E, depends on the order in which transitions are picked on lines 1, 4, and 24 in Figure 1. Fortunately, this does not confuse Tarjan's algorithm, because an edge is introduced either when the algorithm is ready to investigate it or not at all.

In Figure 1, Tarjan's algorithm is represented via lines 2, 8–12, 19, and 25–26. In addition to W, it uses another stack, which we call S. Also its elements are pointers to records in F.

Tarjan's algorithm also uses two attributes on each ω-marking called index and lowlink. The index is a running number that the ω-marking gets when it is encountered for the first time. It never changes afterwards. The lowlink is the smallest index of any ω-marking that is known to belong to the same strong component as the current ω-marking. When backtracking and when encountering an ω-marking that has already been visited and is in the same strong component with the current ω-marking, the lowlink value is backward-propagated and the smallest value is kept. The lowlink value is not backward-propagated from ω-markings that belong to already completed strong components.

Each ω-marking is pushed to S when it is found. It is popped from S when its strong component is ready, that is, the algorithm knows precisely which ω-markings belong to the strong component. After that, the ω-marking never returns to S. Presence in S is tested quickly via an attribute ready that is updated when S is manipulated.

It is easy to check from Figure 1 that an ω-marking is put at the same time to F, A, W, and S, removed from them at different times or not at all, and never again put to any of them. Nothing is removed from F. Always $A \subseteq F$ and $W \subseteq S \subseteq F$. When the algorithm terminates, $W = S = \emptyset$. If W is empty on line 8, then $M = \hat{M}$, and the algorithm terminates without going via line 11. This is why the reference to W.top.lowlink on line 11 is always well-defined. On line 9, the test S.top $\neq W$.top yields true if W is empty but S is not, and false if both are empty.

The following is the central invariant property of Tarjan's algorithm.

Lemma 5.1. Let $M_0 \in F$. There is a path from M_0 to M_c (that is, the M of Figure 1) if and only if $M_0 \in S$. If $M_0 \notin S$, then every ω-marking to which there is a path from M_0 is neither in S nor in W.

Cover-check(M', A) has to find each M_0 such that $M_0 \in A$ and $M_0 < M'$, because they have to be removed from A. When it has found such an M_0, it checks whether $M_0.\mathsf{ready} = \mathsf{false}$, that is, whether $M_0 \in S$. This is a constant-time test that reveals whether there is a path from M_0 to M'. In this way Cover-check detects each holding pumping condition where $M_0 \in A$ with a constant amount of additional effort per removed element of A.

If ω-symbols are added to M', then the checking is started again from the beginning, because the updated M' may cover strictly elements of A that the original M' did not cover strictly. Also they have to be removed and checked against the pumping condition. When Cover-check terminates, there are no holding instances of the pumping condition where $M_0 \in A$, and A no longer contains ω-markings that are strictly covered by M'.

This method only detects the cases where $M_0 \in A$, while [12, 13] use $M_0 \in F$. Fortunately, the next theorem tells that it does not make a difference. We first prove a lemma.

Lemma 5.2. Assume that, unlike in the algorithm, the pumping condition is checked against each ω-marking in F (instead of A). Assume further that $M_0 \xrightarrow{t_1}{}^{\omega} \dots \xrightarrow{t_n}{}^{\omega} M_n$ and $M_n = M_\mathsf{c}$ (where M_c is the current ω-marking). Assume also that i and M'_i satisfy $0 \leq i \leq n$, $M_i \leq M'_i$, and $M'_i \in F$. Then the following hold:

(a) There is a path from M_i to M'_i.

(b) For every $p \in P$, either $M'_i(p) = \omega$ or $M'_i(p) = M_i(p)$.

(c) If $M_\mathsf{c} \in A$, then there is a path from M'_i to M_c.

Proof:

(a) If $M'_i = M_i$, then the claim holds trivially, so let $M_i < M'_i$. Because the algorithm did not reject M_i on line 23, it found M'_i after M_i. When M_i was found, M_i was put into S. Because $M_i \xrightarrow{t_{i+1} \cdots t_n}{}^{\omega} M_\mathsf{c}$, by Lemma 5.1 M_i is still in S. So M_i was in S when M'_i was found. At that moment M'_i became the current ω-marking, so by Lemma 5.1 there was a path from M_i to M'_i. Paths do not disappear, so this path still exists.

(b) If $M'_i = M_i$, then the claim holds trivially, so let $M_i < M'_i$. By it, (a), and the assumption that the pumping condition is checked against F, the pumping condition semi-held with M_i when M'_i was found. So, for every $p \in P$, either $M'_i(p) = M_i(p)$, $M'_i(p)$ was converted to ω, or $M'_i(p)$ was ω to start with.

(c) If the claim does not hold, there is a maximal j and for that j there is a maximal M'_j that satisfy the assumptions of the lemma but there is no path from M'_j to M_c. To derive a contradiction, assume that such j and M'_j exist. Because there is a path from M_j to M_c, we get $M_j \neq M'_j$, so $M_j < M'_j$. Because $M_j < M'_j \in F$, $M_n = M_\mathsf{c} \in A$, and A consists of maximal ω-markings in F, we have $j < n$. So M_{j+1} and the edge $M_j \xrightarrow{t_{j+1}}{}^{\omega} M_{j+1}$ exist. There are two possibilities: t_{j+1} has or has not been tried at M'_j. We derive a contradiction from each one in turn.

(c1) Assume that t_{j+1} has been tried at M'_j. Let the result be called M''_{j+1}.

(c1.1) If M''_{j+1} was kept by the algorithm, there is the edge $M'_j \overset{\omega}{-t_{j+1}\rightarrow} M''_{j+1}$. Consider any pumping condition that held with some $M_\$$ when firing $M_j \overset{\omega}{-t_{j+1}\rightarrow} M_{j+1}$. Because $M_j < M'_j$, and because by (a) there is a path from M_j to M'_j, the condition semi-held with $M_\$$ when firing t_{j+1} at M'_j. So M''_{j+1} has ω-symbols in at least the same places as M_{j+1} has, and we conclude $M_{j+1} \leq M''_{j+1}$. A path from M''_{j+1} to M_c would yield a path from M'_j to M_c, so there is no path from M''_{j+1} to M_c. But then $M''_{j+1} \in F$, $M_{j+1} \leq M''_{j+1}$, and there is no path from M''_{j+1} to M_c, contradicting the assumption that j is the biggest possible.

(c1.2) Assume that M''_{j+1} was rejected by the algorithm. That is, $M'_j\ [t_{j+1}\rangle\ M''_{j+1}$ and when M''_{j+1} was constructed, there was an $M'_{j+1} \in A \subseteq F$ such that $M''_{j+1} < M'_{j+1}$. We show that $M_{j+1} \leq M'_{j+1}$. It clearly holds if $M_{j+1} = M'_{j+1}$, so assume that $M_{j+1} \neq M'_{j+1}$. Because M'_{j+1} did not cause the rejection of the result of firing t_{j+1} at M_j, M'_{j+1} has been found after this firing and thus after M_{j+1}. Because M_{j+1} is still in S, it was in S when M'_{j+1} was found. So by Lemma 5.1 there is a path from M_{j+1} to M'_{j+1}. By Lemma 4.1, if $M_{j+1}(p) = \omega$, then $M'_{j+1}(p) = \omega$. If $M_{j+1}(p) < \omega$, then $M'_{j+1}(p) \geq M''_{j+1}(p) = M'_j(p) + \Delta_{t_{j+1}}(p) \geq M_j(p) + \Delta_{t_{j+1}}(p) = M_{j+1}(p)$. So $M_{j+1} \leq M'_{j+1}$.

Thus M'_{j+1} satisfies the assumptions of the lemma. So there is a path from M'_{j+1} to M_c, because otherwise j would not be the biggest possible. So M'_{j+1} is in S now, was in S when M''_{j+1} was constructed from M'_j, and there is a path from M'_{j+1} to M'_j. By Lemma 4.1, for each $p \in P$, if $M'_{j+1}(p) = \omega$ then $M''_{j+1}(p) = M'_j(p) = \omega$. By (b), each $p \in P$ has either $M'_{j+1}(p) = M_{j+1}(p)$ or $M'_{j+1}(p) = \omega$. Therefore, if $M'_{j+1}(p) < \omega$ then $M'_{j+1}(p) = M_{j+1}(p) \leq M''_{j+1}(p)$. These imply that $M''_{j+1} \geq M'_{j+1}$, which is in contradiction with $M''_{j+1} < M'_{j+1}$.

(c2) Assume that t_{j+1} has not been tried at M'_j. Because there is no path from M'_j to M_c, the algorithm has backtracked from M'_j. Therefore, after M'_j was found but before t_{j+1} was tried at M'_j, an M''_j was found and put to F such that $M'_j < M''_j$, and consequently M'_j.tr was set to nil. By the assumption of the maximality of M'_j, there is a path from M''_j to M_c. So $M''_j \in S$ now and was in S when the algorithm backtracked from M'_j. So there is a path from M''_j to M'_j. By Lemma 4.1, for each $p \in P$, if $M''_j(p) = \omega$ then $M'_j(p) = \omega$. By (b), for each $p \in P$, either $M''_j(p) = M'_j(p) = \omega$ or $M''_j(p) = M_j(p) \leq M'_j(p)$. So $M''_j \leq M'_j$, which is in contradiction with $M'_j < M''_j$.

□

We are ready to prove the main theorem of this section.

Theorem 5.3. The algorithm in Figure 1 constructs the same ω-markings as it would if F were used instead of A in the pumping conditions.

Proof:
In this case the pumping condition is $M' > M_0 \overset{\omega}{-\sigma\rightarrow} M_\mathsf{c}\ [t\rangle\ M_\#$, where $M_0(p) < M'(p) < \omega$ holds for at least one place p, $\sigma = t_1 \cdots t_n \in T^*$, and M' has been made from $M_\#$ by replacing the contents of zero or more places by ω in earlier pumping operations. By Lemma 5.1, from each ω-marking in S and from no ω-marking in $F \setminus S$ there is a path to M_c. The pumping condition triggers the updating of M' to M'' such that for every $p \in P$, either $M_0(p) = M'(p) = M''(p)$ or $M_0(p) < M'(p) \leq M''(p) = \omega$.

We prove the claim by induction. We show that in every pumping operation, A causes (at least) the same updates as F, the induction assumption being that also in the previous times A caused the same updates as F. At least the same updates implies precisely the same updates, because $A \subseteq F$. The induction assumption implies that the data structures have the same contents as they would have if F had been used instead of A. In particular, it justifies the use of Lemma 5.2.

Let the pumping condition hold such that $M_0 \in F$. Lemma 3.2 yields $M_0' \in A$ such that $M_0 \leq M_0'$. We have $M_\mathsf{c} \in A$ because the algorithm is currently trying t at M_c. By Lemma 5.2 (c), there is a path from M_0' to M_c. By Lemma 5.2 (b), for every $p \in P$, either $M_0'(p) = \omega$ or $M_0'(p) = M_0(p)$. If $M_0'(p) = \omega$, then also $M'(p) = \omega$ by Lemma 4.1. So the pumping condition holds with M_0' and causes precisely the same result as M_0 causes. □

6. A Data Structure for Maximal ω-Markings

This section presents a data structure for maintaining A. It has been inspired by Binary Decision Diagrams [3] and Covering Sharing Trees [4]. However, ω-markings are only added one at a time. So we are not presenting a symbolic approach. The purpose of using a BDD-like data structure is to facilitate fast detection of situations where an ω-marking covers another. The details of the data structure have been designed accordingly. We will soon see that they make certain heuristics fast.

We call the M' on line 22 of Figure 1 the *new* ω-marking, while those stored in A are *old*. Cover-check first uses the data structure to detect if M' is covered by any old ω-marking. If yes, then nothing more needs to be done. In the opposite case, Cover-check then searches for old ω-markings that are covered by M'. They are strictly covered, because otherwise the first search would have detected them. The second search cannot be terminated when one is found, because Cover-check has to remove all strictly covered ω-markings from A and use them in the pumping test. Therefore, finding quickly the first old ω-marking that is covered by M' is less important than finding quickly the first old ω-marking that covers M'. As a consequence, the data structure has been primarily optimised to detect if any old ω-marking covers the new one, and secondarily for detecting covering in the opposite order.

Let $\underline{M}(p) = M(p)$ if $M(p) < \omega$ and $\underline{M}(p) = 0$ otherwise. Let $\overline{M}(p) = 1$ if $M(p) = \omega$ and $\overline{M}(p) = 0$ otherwise. In this section we assume without loss of generality that $P = \{1, 2, \ldots, |P|\}$.

The data structure consists of $|P| + 1$ layers. The topmost layer is an array of pointers that is indexed with the total number of ω-symbols in an ω-marking, that is, $\sum_{p=1}^{|P|} \overline{M}(p)$. This number can only be in the range from 0 to $|P|$, so a small array suffices. An array is more efficient than the linked lists used at lower layers. The pointer at index w leads to a representation of the set of ω-markings in A that have w ω-symbols each. The example in Figure 2 has $|P| = 4$. It contains ω-markings with one or two ω-symbols and does not contain ω-markings with zero, three, or four ω-symbols.

Layer $|P|$ consists of $|P| + 1$ linked lists, one for each total number of ω-symbols. Each node v in the linked list number w contains a value $v.m$, a pointer to the next node in the list, and a pointer to a representation of those ω-markings in A that have $\sum_{p=1}^{|P|} \overline{M}(p) = w$ and $\sum_{p=1}^{|P|} \underline{M}(p) = v.m$. The list is ordered in decreasing order of the m values, so that the ω-markings that have the best chance of covering M' come first. Each ω-marking in the example of Figure 2 has either one ω-symbol and a total of six tokens, or two ω-symbols and a total of eight, seven or four tokens.

Let $1 \leq \ell < |P|$. Each node v on layer ℓ contains two values $v.w$ and $v.m$, a link to the next node on the same layer, and a link to a node on layer $\ell - 1$. Of course, this last link is nil if $\ell = 1$. If precisely one path leads to v, then v represents the set of those ω-markings in A that have $\sum_{p=1}^{\ell} \overline{M}(p) = v.w$, $\sum_{p=1}^{\ell} \underline{M}(p) = v.m$, and the places greater than ℓ have the unique ω-markings determined by the path, as will be discussed below. If more than one path leads to v, then v represents more than one subset of

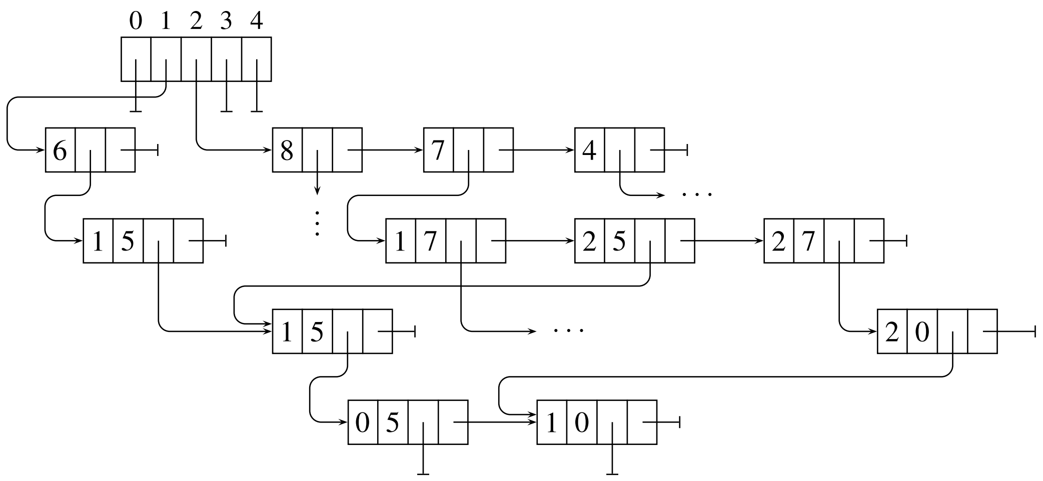


Figure 2. The antichain data structure

A. They are identical with respect to the contents of the places from 1 to ℓ, but differ on at least one place above ℓ. The lists on these layers are ordered primarily in increasing order of the w values and secondarily in increasing order of the m values. Along the leftmost path down in Figure 2, $\sum_{p=1}^{\ell} \overline{M}(p)$ changes as $1, 1, 1, 0$ and $\sum_{p=1}^{\ell} \underline{M}(p)$ as $6, 5, 5, 5$ as ℓ decreases. So the path represents the ω-marking $(5, \omega, 0, 1)$. The other fully shown paths represent $(\omega, 5, 0, 1)$, $(5, \omega, \omega, 2)$, $(\omega, 5, \omega, 2)$, and $(\omega, \omega, 7, 0)$. The node labelled with 2 and 5 represents the set $\{(5, \omega, \omega, 2), (\omega, 5, \omega, 2)\}$.

Like in BDDs, nodes with identical values and next-layer pointers are fused. To be more precise, when a node is being created, it is first checked whether a node with the desired contents already exists, and if yes, it is used instead. A specific hash table makes it fast to find existing nodes based on their contents. The fusion has the consequence that every node that is used to represent M may belong also to the representation of some other ω-marking. So there is no node where the attributes of M could be stored without confusing them with the attributes of other ω-markings. This is why the data structure cannot be used to store the attributes.

Because every ω-marking that is in A is also in F, it has an explicit representation there. As a consequence, unlike with typical applications of BDDs, the storing of dramatically big numbers of ω-markings is not possible. As was mentioned above, the goal is not to do symbolic construction of ω-markings. Even so, the fusing of identical nodes pays off. Otherwise, for each ω-marking, A would use a whole node on layer 1 and additional partially shared nodes on other layers, while F represents the ω-marking as a dense vector of bytes. So A would use much more memory for representing each ω-marking than F uses. This is apparent in the measurements reported in Section 8. Another optimisation facilitated by node fusion will be mentioned later in this section.

Consider the checking whether $M' \leq M$, where M' is the new and M is any old ω-marking. At the topmost layer, all entries i with $|P| \geq i \geq \sum_{p=1}^{|P|} \overline{M}'(p)$ are tried (unless the algorithm stops earlier because of finding coverage).

After entering a node v at level $\ell - 1$, $M(\ell)$ is computed as $u.w - v.w$ and $u.m - v.m$, where u is the node at level ℓ from which level $\ell - 1$ was entered. If $M'(\ell) > M(\ell)$, then the traversal backtracks to u. This is because, thanks to the ordering of the lists, both v and the subsequent nodes in the current list at level $\ell - 1$ correspond to ω-markings whose $M(\ell)$ is smaller than $M'(\ell)$.

On the other hand, M may be rejected also if it has fewer ω-symbols in the so far uninvestigated places than M' has, that is, if $\sum_{p=1}^{\ell-1} \overline{M}'(p) > \sum_{p=1}^{\ell-1} \overline{M}(p)$. To quickly detect this condition, an array wsum is pre-computed such that $\mathsf{wsum}[\ell] = \sum_{p=1}^{\ell} \overline{M}'(p)$:

```
$\mathsf{wsum}[1] := \overline{M}'(1)$
**for** $p := 2$ **to** $|P|$ **do** $\mathsf{wsum}[p] := \mathsf{wsum}[p-1] + \overline{M}'(p)$ **endfor**
```

This pre-computation introduces negligible overhead. The condition becomes $\mathsf{wsum}[\ell-1] > v.w$, which is a constant time test. If this condition is detected, layer $\ell - 2$ is not entered from v, but the scanning of the list on layer $\ell - 1$ is continued.

The current node v (but not the current list) is rejected also if $\mathsf{wsum}[\ell-1] = v.w$ and $\mathsf{msum}[\ell-1] > v.m$, where $\mathsf{msum}[\ell] = \sum_{p=1}^{\ell} \underline{M}'(p)$ is a second pre-computed array. This is because of the following. Consider the uninvestigated places $p \in \{1, \ldots, \ell - 1\}$. If some of them has $M(p) < \omega = M'(p)$, then it is correct to reject M. Otherwise $M(p) = \omega$ if and only if $M'(p) = \omega$, but M does not have altogether enough tokens in the uninvestigated non-ω places to cover M'.

There also is a third pre-computed array mmax with $\mathsf{mmax}[\ell]$ being the maximum of $\underline{M}'(1)$, $\underline{M}'(2)$, $\ldots$, $\underline{M}'(\ell)$. It is used to reject v when

$$\mathsf{wsum}[\ell-1] < v.w \ \text{ and } \ (v.w - \mathsf{wsum}[\ell-1]) \cdot \mathsf{mmax}[\ell-1] + v.m < \mathsf{msum}[\ell-1] \ .$$

The idea is that M has more ω-symbols in places $1, \ldots, \ell - 1$ than is needed to cover the ω-symbols in M' in the same places. Each extra ω-symbol in M in those places covers at most $\mathsf{mmax}[\ell-1]$ ordinary tokens in M' in those places. If the test triggers, the extra ω-symbols together with the ordinary tokens in M in those places do not suffice to cover the ordinary tokens in M' in those places.

There is thus a fast heuristic for each of the cases $\mathsf{wsum}[\ell-1] < v.w$, $\mathsf{wsum}[\ell-1] = v.w$, and $\mathsf{wsum}[\ell-1] > v.w$. These heuristics are the reason for storing $\sum_{p=1}^{\ell} \overline{M}(p)$ and $\sum_{p=1}^{\ell} \underline{M}(p)$ into the node instead of $M(\ell)$.

Consider the situation where none of the above heuristics rejects v. Then layer $\ell - 2$ is entered from v. If it turns out that M' is covered, then the search need not be continued. In the opposite case, it is marked into v that layer $\ell - 2$ was tried in vain. If v is encountered again during the processing of the same M', this mark is detected and v is not processed further. Encountering again is possible because of node fusion.

To avoid the need of frequently resetting these marks, the mark is a running number that is incremented each time when the processing of a new M' is started. The marks are reset only when this running number is about to overflow the range of available numbers. This trick is from [12, 13].

Heuristics that are similar to the first two are used for checking whether the new ω-marking strictly covers any ω-marking in A. The biggest difference is that now the search cannot be stopped when such a situation is found, as has been explained above. The condition "strictly" need not be checked, because if $M' \in A$, then M' is rejected by the first search or already on line 20. The third heuristic mentioned above is not used, because a value that corresponds to $\mathsf{mmax}[\ell]$ cannot be implemented cheaply for ω-markings in A. Storing the value in the node would require an extra field in the record for the nodes, increasing memory consumption. Not storing it would imply time-consuming repeated re-computation of it. Furthermore, a node may represent more than one $(M(1), M(2), \ldots, M(\ell))$. For correctness, the maximum of their $\mathsf{mmax}[\ell]$ values must be used, reducing the power of the idea. For instance, the node

```
 ⋮   ⋮
 3   while $W \neq \emptyset$ do
 ⋮   ⋮
 6           $W$.pop
 7           activate transitions as discussed in Section 7
 ⋮   ⋮
16       $M'$ := the $\omega$-marking such that $M\,[t\rangle\,M'$
17       if $M' \leq M$ then passivate $t$; go to line 3 endif
 ⋮   ⋮
```

Figure 3. Adding transition removal optimisation

labelled with 7 on layer 4 in Figure 2 has three fully shown paths down, with mmax$[4]$ values of 5, 5, and 7. The value stored in the node would have to be 7.

7. Transition Removal Optimisation

If the firing of a transition t does not increase the ω-marking of any place, that is, if $M\,[t\rangle\,M'$ and $M' \leq M$, then t is *useless*. Lines 18 and 20 or 22 and 23 in Figure 1 would reject M', had it not already been done on line 17. Line 19 would not have any effect, because if the assignment on it is executed, then $M' = M$. This is because by $\neg M'$.ready, there is a path from M' to M. So, for each $p \in P$, either $M'(p) = M(p)$ or $M'(p) < M(p) = \omega$ ($M' \leq M$ rules out $M'(p) > M(p)$). However, $M'(p) < M(p) = \omega$ is impossible because of $M\,[t\rangle\,M'$.

A transition that is not originally useless in this sense becomes useless, if ω-symbols are added to each p such that $W(p,t) < W(t,p)$. By Lemma 4.1, it remains useless until the algorithm backtracks beyond where it became useless.

That t has become useless is easy and cheap to detect immediately after firing it, by checking that the resulting ω-marking M' is covered by the current ω-marking M. However, M is not necessarily the ω-marking M_{u} where t became useless. It may be that before trying t at M_{u}, the algorithm tries other transitions, proceeds to some other ω-marking, and detects there that t has become useless. That is, the ω-marking where t became useless is not necessarily the current top of the stack W. Instead, it is one of those ω-markings in W where ω-symbols were added to the ω-marking. (Alternatively, t has been useless to start with.)

Lines 7 and 17, shown again in Figure 3, implement an additional optimisation based on these facts. The "first transition" and "next transition" operations in Figure 1 pick the transitions from a doubly linked list which we call the *active list*. Line 17 tests whether the current transition t has become useless. If yes, then t is linked out from the active list and inserted to a singly linked list that starts at passive$[c]$, where c is the number of locations in W where ω-symbols have been added, and passive is an array of size $|P| + 1$. There also is a similarly indexed array toW such that toW$[c]$ points to the most recent location in W where ω-symbols have been added. The forward and backward links of t still point to the earlier successor and predecessor of t in the active list. This operation takes constant time.

From then on, t is skipped at no additional cost until the algorithm backtracks to the nearest ω-marking where ω-symbols were added. This moment is recognized from the current top of W getting below toW$[c]$. Then all transitions from passive$[c]$ are removed from there and linked back to their original places in the active list, and c is decremented. Because each passive list is manipulated only at the front, it releases the transitions in opposite order to in which they were inserted to it. This implies that the original ordering of the active list is restored when transitions are linked back to it, and the "next transition" operation is not confused. Also the linking back is constant time per transition.

If the check on line 17 were removed, the algorithm would still reject M', but in the worst case that might happen much later in Cover-check. This heuristic is very cheap and may save time by rejecting M' early. Unfortunately, in our experiments (Section 8) it did not have a significant effect to the running time in either direction. For the time being, the evidence suffices neither for adopting nor for rejecting this optimisation.

8. Experiments and Conclusions

In this section we present some experiments that we have made with an implementation of our new algorithm. Before going to them, let us discuss the general nature of this kind of measurements a bit.

With many algorithms, knowing the running times of some implementation with some set of inputs on some computer makes it possible to estimate what the running time of the same implementation would be with some other input on the same computer. Unfortunately, as was mentioned in Section 1, this seems to not be the case with coverability set construction. Therefore, the numbers reported in this section must not be considered indicative of the performance on Petri nets of the same size, and not even on the very same Petri net. Also the ordering in which the transitions of the Petri net are given must be the same. No systematic assessment of the effect of the ordering was made for this section. The ad-hoc experiments mentioned towards the end of this section indicate that the effect is there.

Furthermore, it is a repeatedly observed experimental fact that the running time of precisely the same program on precisely the same computer and operating system with precisely the same input on the very same day varies, even if Internet connection has been switched off, the computer is always running on mains electricity, and the battery is fully charged. To make it possible to assess this effect, in this section the fastest and slowest of five identical measurements are reported.

To make comparison of running times reasonable to the extent possible, we compare the new implementation to a slightly improved version of the implementation in [12, 13] (better hash function, etc.). Both have been written in the same programming language (C++) and were executed on the same computer. The new implementation was derived from the earlier implementation. In particular, for literally identical input, they use the same ordering of the transitions.

The Petri net mesh2x2 is the heaviest example from [7, 10]. It has been included to point out that the examples from [7, 10] are not challenging enough for testing the new implementation. Mesh3x2 is a bigger version of it. AP13a has been modified from users.cecs.anu.edu.au/~thiebaux/benchmarks/petri/ by Henri Hansen, to present a somewhat bigger challenge. SmallS5x2 was designed for this publication, to have a small S and offer many possibilities for fusing nodes in the representation of A. LargeS2x15x2 had precisely the opposite design goal with respect to S.

The rest of the tested models are from the Model Checking Contest @ Petri Nets 2014, later MCC. Of these the net SharedMemory-10 has also been tested in [2], where the reachability graph of the Petri

Table 1. Some measurements with the new algorithm and its variants

	mesh2x2			mesh3x2			AP13a		
$\lvert A\rvert$ $\lvert F\rvert$ $\lvert S\rvert$	256	316	316	6400	7677	7677	1245	1281	65
$\approx$ [12, 13]	4	5	23	696	739	718	60	66	467
no node fusion	2	2	70	46	48	1389	49	56	4393
basic new	3	4	24	52	68	732	57	62	850
no tr. removal	4	4	24	64	76	732	56	60	850
partial F	3	4	22	55	61	672	54	71	401

	smallS5x2			largeS2x15x2			SharedMemory-10		
$\lvert\ \rvert$	31752	31752	50	32768	32768	32768	1830519	1830519	1830519
$\approx$	3151	3182	1736	7121	7142	3328	timeout		
nnf	67	72	4058	231	237	20736	20553	20556	600738
bn	80	89	1738	248	273	4224	28022	28064	313033
ntr	81	91	1738	261	270	4224	28200	28327	313033
pF	152	162	4	250	258	3968	27144	27192	298732

	GlobalResAlloc-3			ResAllocation-R-15			ResAllocation-C-10		
$\lvert\ \rvert$	4096	4355	4355	278528	278528	65440	823552	823552	205920
$\approx$	665	673	331	504573	505378	28288	timeout		
nnf	510	515	1080	37226	37353	186240	12516	12452	288533
bn	547	559	334	12159	12195	29569	15086	15094	83832
ntr	515	519	334	12266	12280	29569	15198	15222	83832
pF	530	538	300	27021	27058	7416	31301	31368	19495

net was constructed using a cluster of computers. We also tried to run SimpleLoadBal-10 and Planning from MCC, but all versions of the algorithm ran out of memory. As stated in [2], the reachability graph of SimpleLoadBal-10 is so large it goes beyond the capabilities of a single machine.

The results are in Table 1. For each example, the second row shows the final sizes of the sets A and F and the maximal size of the stack S, excluding "partial F" (discussed soon). The sizes of W are not shown, because always $|W| \leq |S|$.

The next five rows show results for various implementations.

- "$\approx$ [12, 13]" or "$\approx$" was explained above.
- "Basic new" or "bn" is the algorithm described in this publication.
- "No node fusion" or "nnf" is otherwise the same as "basic new", but nodes in the data structure for A that have the same values and next-layer pointers are not fused. Its results are shown above "basic new" to present the new implementations in the order of decreasing memory consumption.
- "No tr. removal" or "ntr" is otherwise the same as "basic new", but the optimisation discussed in

Section 7 is not in use.

- "Partial F" or "pF" is otherwise the same as "basic new", but when an ω-marking is removed from S it is also removed from F (but not from A). As a consequence, the final F is empty, and the maximal size of F is $|S|$. This heuristic is correct because if a removed ω-marking is constructed anew, it is covered by some ω-marking in A, and thus will be rejected on line 23 at the latest. The goal is to save memory at the cost of spending more time. As long as an ω-marking is in S, it cannot be removed from F, because its lowlink and index values are needed, and they are stored in F.

For each implementation and Petri net, the first two numbers report the shortest and longest running times for five identical measurements in milliseconds. A timeout was set at one hour. The third number is the amount of memory consumed, measured in kibibytes, assuming that memory is reserved for A, S, and the base table of F only as needed. (According to ISO/IEC 80000, the prefix that denotes 1024 is "kibi". The use of "kilo" for 1024 should be avoided.) For W, the same amount of memory was reserved as for S. These numbers are theoretical in the sense that the implementation did not use dynamically growing arrays in reality.

To protect against programming errors, we checked for each Petri net that every version returned the same A. In [2], 1.831×10^6 markings were obtained for SharedMemory-10, which matches our figure 1 830 519.

All new versions are significantly faster than the one in [12, 13] excluding AP13A and GlobalRes-Alloc-3 where all versions are roughly equally fast, and mesh2x2 that is too small for a meaningful comparison. Although precise comparison is not possible, the results on mesh2x2 make it obvious that the new implementation outperforms the one in [10]. In [2], SharedMemory-10 was analysed in a parallel environment ranging from 2 to 16 machines. The time consumption ranged from 5 hours 25 minutes to 1 hour 14 minutes, compared to our results that are between 20 and 29 seconds. On the other hand, our algorithm failed but [2] succeeded with SimpleLoadBal-10. That is, compared to [2], our algorithm seems to be very competitive in terms of time consumption on models which are small enough to be run on a single machine.

The running times present a surprise in the model ResAllocation-R-15, where "no node fusion" is the slowest of the new versions. We leave an analysis of this phenomenon as potential future work.

The memory consumptions of the four new versions relate to each other as one would expect. Compared to [12, 13] whose A was a doubly linked list of the same records that F used, the new versions consume much more memory except when the fusion of nodes and the removal of ω-markings from F have a big effect. While [12, 13] use two additional pointers per ω-marking to represent A, the new versions have the complicated structure described in Section 6. Furthermore, S was absent from [12, 13].

For some Petri net / algorithm version combinations we also tried giving the transitions in a reversed order, which resulted in varying outcomes. The running time shortened, stayed the same or lengthened, sometimes even tripled. We did not observe changes in the relative ordering of the different versions, though it might be possible in some cases.

Acknowledgements. We thank the anonymous reviewers of both RP '13 and Fundamenta Informaticae for their effort. In particular, the comments by the latter led to significant improvements to the presentation of the main ideas.

References

[1] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA (1974)

[2] Bellettini, C., Camilli, M., Capra, L., Monga, M.: MaRDiGraS: Simplified Building of Reachability Graphs on Large Clusters. In: Abdulla, P.A., Potapov, I. (eds.) Reachability Problems, 7th International Workshop, LNCS, vol. 8169, pp. 83–95. Springer (2013)

[3] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C–35 (8), 677–691 (1986)

[4] Delzanno, G., Raskin, J.-F., Van Begin, L.: Covering Sharing Trees: A Compact Data Structure for Parameterized Verification. Software Tools for Technology Transfer 5(2-3), 268–297 (2004)

[5] Eve, J., Kurki-Suonio, R.: On Computing the Transitive Closure of a Relation. Acta Informatica 8(4), 303–314 (1977)

[6] Finkel, A.: The Minimal Coverability Graph for Petri Nets. In: Rozenberg, G. (ed.) Advances in Petri Nets 1993, LNCS, vol. 674, pp. 210–243. Springer (1993)

[7] Geeraerts, G., Raskin, J.-F., Van Begin, L.: On the Efficient Computation of the Minimal Coverability Set of Petri Nets. International Journal of Foundations of Computer Science 21(2), 135–165 (2010)

[8] Karp, R.M., Miller, R.E.: Parallel Program Schemata. Journal of Computer and System Sciences 3(2), 147–195 (1969)

[9] Piipponen, A., Valmari, A.: Constructing Minimal Coverability Sets. In: Abdulla, P.A., Potapov, I. (eds.) Reachability Problems, 7th International Workshop, LNCS, vol. 8169, pp. 183–195. Springer (2013)

[10] Reynier, P.-A., Servais, F.: Minimal Coverability Set for Petri Nets: Karp and Miller Algorithm with Pruning. Fundamenta Informaticae 122(1-2), 1–30 (2013)

[11] Tarjan, R.E.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)

[12] Valmari, A., Hansen, H.: Old and New Algorithms for Minimal Coverability Sets. In: Haddad, S., Pomello L. (eds.) PETRI NETS 2012, LNCS, vol. 7347, pp. 12–21. Springer (2012)

[13] Valmari, A., Hansen, H.: Old and New Algorithms for Minimal Coverability Sets. Fundamenta Informaticae 131(1), 1–25 (2014)