Universiteit Leiden
Opleiding Informatica

Comparing algorithms:

calculating the minimal coverability set of Petri nets

Name:	Oscar Brandt
Date:	February 15, 2017
1st supervisor:	Jetty Kleijn
2nd supervisor:	Hendrik Jan Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Comparing algorithms:

calculating the minimal coverability set of Petri nets

Oscar Brandt

Abstract

There are several algorithms for computing the Minimal Coverability Set of a Petri net. This thesis compares the Karp&Miller approach, the faulty MCG algorithm by Finkel, and three newer algorithms. The newer algorithms sacrifice some speed for correctness, and the details of the algorithms are discussed. The algorithms are implemented in C#, and are then run on a selection of Petri nets, to compare the results.

Acknowledgements

I want to thank my supervisor Jetty Kleijn for her endless patience.

The same goes for my parents, and Manon.

Contents

List of Figures

Chapter 1

Introduction

This thesis will elaborate on the performance of some algorithms for finding the Minimal Coverability Set of a Petri net. Chapter 2 explains the definitions of Petri nets and Coverability sets we will use. Coverability sets are finite representations of infinite sets using ω tokens to mean infinite supply, and are created by exploring the set of reachable nodes until the *pumping lemma* can be used. The pumping lemma uses the *strict monotonicity* of Petri nets to determine when an infinite number of markings are reachable. Usage of the lemma is reliant on the order of exploration, so multiple algorithms for calculating the Minimal Coverability set exist, with various optimisations.

Several papers are reviewed in Chapter 3, containing algorithms for calculating the Minimal Coverability Set, and their algorithms are briefly explained: the Karp&Miller tree, the MCG algorithm, the CoverProc algorithm, the Monotone-Pruning algorithm, and the unnamed algorithm by Hansen and Valmari. The properties of these algorithms are discussed in more detail in Chapter 4. Chapter 5 holds the empirical evaluation; all algorithms were implemented in C#, and executed on various Petri nets. Several graphs give a quick impression of their performance.

The appendix A contains some of the program code used, and appendix B has some as visual representations of the Petri nets used to compare the algorithms.

Chapter 2

Preliminaries

2.1 Petri nets

Definition 2.1. A *Petri net graph* is a mathematical model that is represented as a directed bipartite graph. A Petri net graph is a tuple (P, T, W) of a set of *places* P, a set of *transitions* T, and a *weight* function $W : (P \times T) \cup (T \times P) \mapsto \mathbb{N}$. The sets P and T are disjoint, an object can not be both a place and a transition. In this report the sets P and T are finite. W is often represented as a multiset of arcs, where a mapping $W(p,t) = n > 0$ means an arc of weight n from p to t, a mapping $W(t,p) > 0$ is an arc from t to p, and a mapping to 0 is the absence of an arc.

Definition 2.2. The *input of a transition* $t \in T$, denoted ${}^\bullet t$, is the set of places so that $\forall p \in {}^\bullet t : W(p,t) > 0$, and the output $t^\bullet$ is the set of places so that $\forall p \in t^\bullet : W(t,p) > 0$. Simpler said, the sets of places with an arc to/from t. These two functions need not be 'disjunct', there can be an arc between a place and a transition in both directions.

Likewise, the *input* ${}^\bullet p$ *of a place* $p \in P$ is the set of transitions with an arc to p, and the output $p^\bullet$ is the set of transitions with an arc from p.

Definition 2.3. A *marking* M (or *configuration* C) of a Petri net graph is a function $M : P \mapsto \mathbb{N}$ that assigns an integer value to each place, denoting its *supply*.

We can represent a marking just as easily with a vector of $|P|$ elements of $\mathbb{N}$, as long as it is known what the order of the places is, or as a multiset of the set of places. Choosing the vector representation, addition and subtraction work naturally: let A, B and C be markings, then $A = B + C \Leftrightarrow \forall p \in P : A(p) = B(p) + C(p)$, and $A = B - C \Leftrightarrow \forall p \in P : A(p) = B(p) - C(p)$.

If a marking contains a lot of empty places and the order of P is obvious, we can instead represent it with

a string denoting only the positive places, and their supply if it is more than 1. For example, $\langle 0, 1, 2, 0 \rangle = \{p_2, 2p_3\}$. We still often use the function notation $M(p)$ to denote the supply at a specific place.

Definition 2.4. The partial order $\leqslant$ works naturally, where $M \leqslant M'$ means $M(p) \leqslant M'(p)$ for every $p \in P$, and we say that M' *covers* M. If furthermore $M \neq M'$, we say that $M < M'$, and M' *strictly covers* M. If neither marking covers the other, they are *incomparable*.

Definition 2.5. A set $\mathcal{M}$ of markings *covers* a set $\mathcal{N}$ of markings if for every marking $N \in \mathcal{N}$ there exists a marking $M \in \mathcal{M}$ so that $N \leqslant M$.

A *marked Petri net graph* is a tuple (P, T, W, M) where (P, T, W) is a Petri net graph and M is a marking for the places P.

Definition 2.6. A *Petri net*, also known as a *place/transition net* or *P/T net* is a tuple $PN = (P, T, W, \hat{M})$ where (P, T, W) is a Petri net graph, and $\hat{M}$ (often called M_0) is the *initial marking* representing the starting state of the system. It is usually represented as a diagram like Figure 2.1, where

- places P are circles
- transitions T are rectangles and
- the weight function W is depicted as arrows between the former two, with a number next to the arrow denoting the weight of the arc. If the number is omitted, the arc has a weight of 1.
- the supply in each place p, $M(p)$, is the amount of dots within that place's circle.

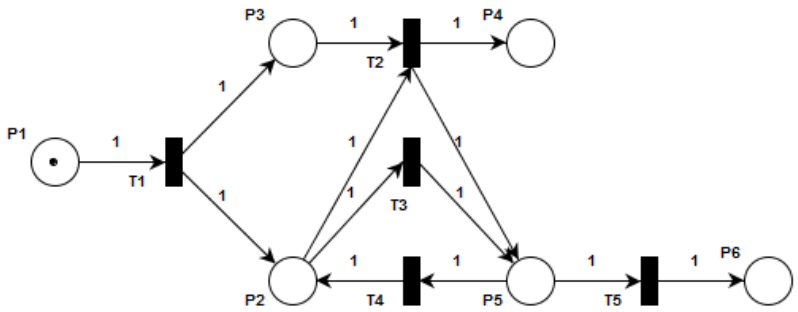


Figure 2.1: A basic Petri net.

Definition 2.7. A transition t is *enabled at* M, denoted with $M\,[\,t\,\rangle\,\cdot$, if and only if $M(p) \geqslant W(p, t)$ for every $p \in P$. Then t may *fire*, 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 $M\,[\,t\,\rangle\,M'$. The *effect of* t *on* p is $t(p) = W(t, p) - W(p, t)$.

The notation is extended to sequences of transitions in a natural recursive way: Let $t \in T$, $\sigma \in T^+$, then $M\,[\,t\sigma\,\rangle\,M''$ if and only if $M\,[\,t\,\rangle\,M'$ and $M'\,[\,\sigma\,\rangle\,M''$. Also, we use $M\,[\,\sigma^i\,\rangle\,M'$ to mean the marking M' is produced by firing σ from M in succession i times, e.g. $M\,[\,\sigma^3\,\rangle\,M' \Leftrightarrow M\,[\,\sigma\sigma\sigma\,\rangle\,M'$

Lemma 1. Petri nets are *strictly monotonic*: Let M_0, M_1 s.t. $M_0 < M_1$, and $\sigma \in T^*$ s.t. $M_0 [\sigma\rangle M_0'$. Then $M_1 [\sigma\rangle M_1'$ and $M_0' < M_1'$.

Definition 2.8. 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* (or *reachability set*)$\mathcal{R}$ of a Petri net is the set of markings reachable from the initial marking $\hat{M}$.

Definition 2.9. A transition t in a Petri net PN is *live* if for every reachable marking $M \in \mathcal{R}, \exists \sigma \in T^* : M [\sigma t\rangle \cdot$. A Petri net is *live* if every transition in it is live.

We can visualize the set of reachable markings with a *reachability tree* RT. A reachability tree (N, E, Λ) of a Petri net is an edge-labelled directed rooted tree, with nodes N, labelled edges E, a labelling $\Lambda : N \mapsto \mathbb{N}^{|P|}$ and a root labelled $\hat{M}$. For every node n, for every transition t enabled at the node's label M s.t. $M [t\rangle M'$, there is one arc labelled t from n to a node n' labelled M'. We call n the *parent* of n', and n' the *child* of n. The nodes with a path to n are the *ancestors* of n, and the nodes with a path from n are the *descendants* of n. The set of markings found in a reachability tree is thus the reachability set.

Example 1. Figure 2.2 is the reachability tree of Figure 2.1, built by recursively exploring all transitions possible in each node and creating new nodes connected to it. The problem is already apparent, *cycles* make this an infinite tree. We can identify nodes with equal markings with each other to create a *reachability graph* RG (or sometimes Sequential Configuration Graph SCG) instead, as in Figure 2.3.

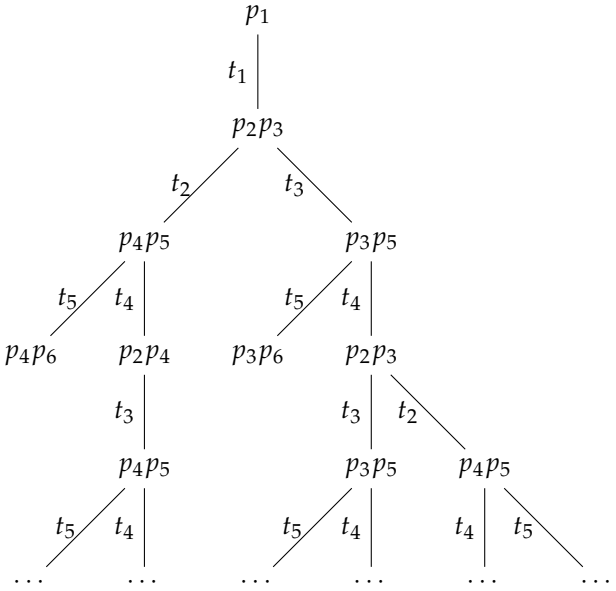


Figure 2.2: An infinite reachability tree of Figure 2.1.

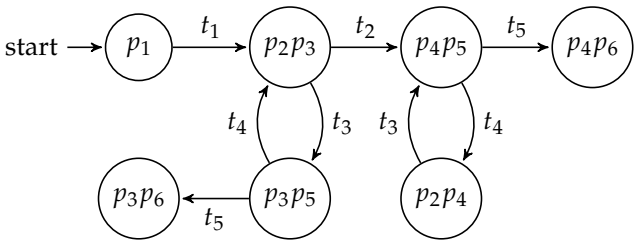


Figure 2.3: A reachability graph of Figure 2.1.

2.2 ω-markings

Now consider the Petri net in Figure 2.4. While still a simple net, there is a problem in constructing the reachability graph: Let $M_0 = \hat{M} = \{1, 0, 0\}$. $M_0 \,[\, t_1 \,\rangle\, M_0' = \{0, 1, 0\}$, and $M_0' \,[\, t_2 \,\rangle\, M_1 = \{1, 0, 1\}$. Because of the monotonicity of Petri nets, we can fire the sequence $t_1 t_2$ again, to get $M_2 = \{1, 0, 2\}$. In fact, there is an infinite amount of reachable markings, by repeatedly firing $t_1 t_2$: $M_0 \,[\, (t_1 t_2)^n \,\rangle\, M_n$, where $M_n = \{1, 0, n\}$, for every $n \in \mathbb{N}$. The set of reachable markings is infinite, and because constructing the reachability graph is exploring the set of reachable markings, an algorithm that constructs the reachability graph might not terminate.

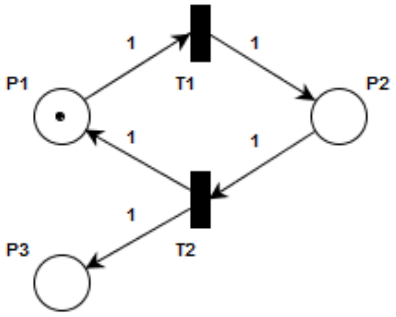


Figure 2.4: An *unbounded* Petri net.

Definition 2.10. A place in a Petri net is *unbounded* if $\forall n \in \mathbb{N}, \exists M \in \mathcal{R} : M(p) \geqslant n$. A Petri net is *unbounded* if any of its places are unbounded. An unbounded Petri net has an infinite reachability set, tree, and graph.

Lemma 2. The pumping lemma. Let M and M' be markings, $\sigma \in T^+$, $M \,[\, \sigma \,\rangle\, M'$ and $M < M'$. Then there exist an infinite amount of markings $M_0 = M, M_1 = M', M_2, \ldots$ where $M \,[\, \sigma^n \,\rangle\, M_n$, for every $n \in \mathbb{N}$.

Definition 2.11. To more easily talk about unbounded places, we introduce the *ω-marking*. An ω-marking is a vector of $|P|$ elements of $\mathbb{N} \cup \{\omega\}$, where ω means "unbounded", a place with an infinite supply. $\forall n \in \mathbb{N} : n < \omega$, and $n + \omega = \omega = \omega + n = \omega - n$. We use $\Omega(M) = \{p \mid M(p) = \omega\}$ and $\overline{\Omega}(M) = \{p \mid M(p) \neq \omega\}$.

A transition t is enabled in an ω-marking M if and only if $M(p) \geqslant W(p, t)$ for every $p \in P$. Then t may *fire*, yielding the ω-marking M' such that

$$M'(p) = \begin{cases} \omega & \text{if } M(p) = \omega \\ M(p) - W(p,t) + W(t,p) & \text{otherwise} \end{cases}$$

Just as with standard markings, we can extend this notation to sequences of transitions. The notions of monotonicity, reachability and the pumping lemma, apply to ω-markings as well.

2.3 Coverability Sets

To analyse the behaviour of a Petri net, we are often interested in the set of reachable markings $\mathcal{R}$. For finite sets this is simple enough, but if $\mathcal{R}$ is infinite we instead try to find a set that covers $\mathcal{R}$. Consider the net in Figure 2.4. No finite set of standard markings covers $\mathcal{R}$, for there are always markings $\{1, 0, x\}$ and $\{0, 1, x\}$, $x \in \mathbb{N}$ not covered, as the third place is unbounded. Instead, the set $\{\{1, 0, \omega\}, \{0, 1, \omega\}\}$ suffices, as any marking is covered by either element.

We could also say $\mathcal{R}$ is covered by $\{\omega, \omega, \omega\}$, but that's not so useful. Even if all the places with an ω are unbounded this is a bad idea: consider the Petri nets below. p_3 and p_6 are unbounded in both, but only one of them can be used during a single **execution** of the top net.

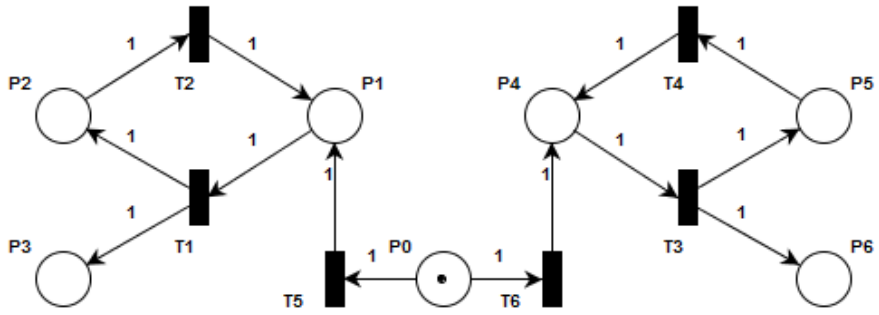


Figure 2.5: A Petri net with two unbounded places.

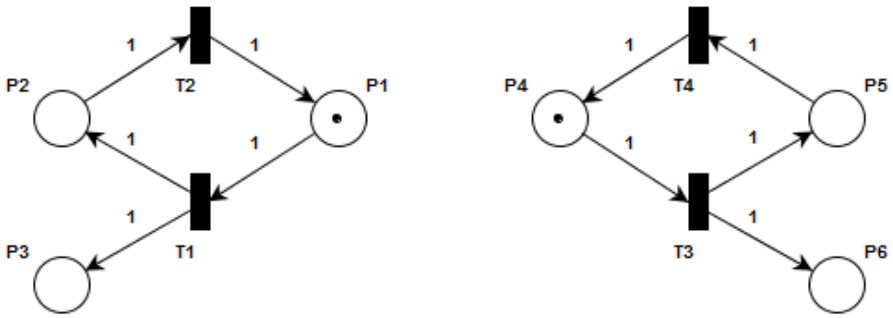


Figure 2.6: A Petri net with two simultaneously unbounded places.

Definition 2.12. We say that an ω-marking M is a *limit* of a set $\mathcal{M}$ of ω-markings if and only if $\mathcal{M}$ contains an infinite sequence (M_n) so that $M_0 \leqslant M_1 \leqslant M_2 \leqslant \ldots$ and $\forall n \in \mathbb{N}$, $\forall p \in P$

- either $M(p) \neq \omega$ and $M_n(p) = M(p)$,

- or $M(p) = \omega$ and $M_n(p) \geqslant n$.

Intuitively, this means that ω-symbols are in a limit if and only if that place is 'unbounded in $\mathcal{M}$'. We can also say that M is a limit of the sequence (M_n). In particular, by definition, a limit of a set covers all the elements of its infinite sequence. Using this, we can extend the notion of a set covering $\mathcal{R}$ to ω-markings:

Definition 2.13. A *coverability set* (or *covering set*) CS of a Petri net is any set of ω-markings $\mathcal{M}$ that satisfies:

(i) Every reachable marking M is covered by some $M' \in \mathcal{M}$.

(ii) Every $M' \in \mathcal{M}$ that is not in $\mathcal{R}$ is a limit of $\mathcal{R}$.

The set of reachable markings is a coverability set, as every marking is covered by itself, and there are no elements in $\mathcal{M}$ that are not in $\mathcal{R}$.

Theorem 2.1. Every unbounded Petri net has a finite coverability set.

Example 2. Consider the Petri net in Figure 2.4. The set $\{\{1,0,\omega\},\{0,1,\omega\}\}$ is a coverability set of this graph, as every reachable marking is covered by either of these markings, and they are limits of the sequences of reachable markings $(M_1)_n = [\{1,0,0\},\{1,0,1\},\{1,0,2\},\dots]$ and $(M_2)_n = [\{0,1,0\},\{0,1,1\},\{0,1,2\},\dots]$.

The set $\{\{1,0,\omega\},\{0,1,\omega\},\{1,0,0\}\}$ is a coverability set as well. The set $\{\{1,1,\omega\}\}$ is not a coverability set, as that marking is not a limit, there would need to be at least one reachable marking of form $\{1,1,x\}$.

Example 3. A coverability set for the Petri net in figure 2.5 is $\{\{1,0,0,0,0,0,0\},\{0,1,0,\omega,0,0,0\},\{0,0,1,\omega,0,0,0\},\{0,0,0,0,1,0,\omega\},\{0,0,0,0,0,1,\omega\}\}$. Note that no marking with an ω in both p_3 and p_6 can be in the coverability set; we can not construct a sequence of reachable markings that grows arbitrarily large in both places at the same time. A coverability set for the Petri net in figure 2.6 is $\{\{1,0,\omega,1,0,\omega\},\{1,0,\omega,0,1,\omega\},\{0,1,\omega,1,0,\omega\},\{0,1,\omega,0,1,\omega\}\}$. Just markings with a single ω will not suffice, there would be reachable markings not covered by any element of the set.

From this, we can conclude that a coverability set is a good substitute for $\mathcal{R}$. $\mathcal{R}$ is a coverability set itself, and there are different sets for different behaviour. We can visualise a coverability set CS with its *coverability graph* CG(CS) of a net PN. It is the graph so that its nodes are in one-to-one correspondence with the coverability set, and arcs are between nodes M and M' labelled t if and only if $M\,[\,t\,\rangle\,M'$. Both nodes need to be in the coverability set, so this construction ensures a graph unique to each coverability set, even if some nodes do not have any arcs.

Definition 2.14. A coverability set CS is *minimal* if no proper subset of CS is a coverability set.

Lemma 3. All elements in a minimal coverability set are incomparable to each other.

Proof. Let $\mathcal{M}$ be a coverability set, $M, M' \in \mathcal{M}$ and $M < M'$. Then $\mathcal{M} - \{M\}$ is a coverability set too, so $\mathcal{M}$ was not minimal. □

Lemma 4. *(Dickson's Lemma)* [Dic13] The $\leqslant$ relation on markings is a well partial order, i.e.: Every infinite sequence of markings has an infinite increasing subsequence.

Lemma 5. Minimal coverability sets are finite.

Proof. Let $\mathcal{M}$ be an infinite coverability set. Group the elements of $\mathcal{M}$ by their sets of places with an ω token, at least one such group has an infinite number of elements. Regard this group as a sequence. Disregarding the ω tokens, we can use Dickson's Lemma to state that there is an infinite increasing subsequence of these elements. As these elements are distinct, there are comparable elements in $\mathcal{M}$, and it can not be a minimal coverability set. □

Theorem 2.2. There is a unique *minimal coverability set* MCS.

This has already been proven [Fin91], but as the definitions of coverability sets are often equivalent but not equal, we shall prove it here:

Proof. Let $\mathcal{M}$ and $\mathcal{N}$ both be minimal coverability sets of some Petri net, $M \in \mathcal{M}$.

1) $M \in \mathcal{R}$. There is a marking $N \in \mathcal{N}$covering M.

   - $N \in R$. As $N \in \mathcal{R}$, there must also be a marking $M' \in \mathcal{M}$ covering N. $M \leqslant N \leqslant M'$ implies $M = N = M'$ by minimality of $\mathcal{M}$.
   - $N \notin R$. N is the limit of the sequence of reachable markings (N_n). Let $i = \max_{p \in \Omega(N)} M(p)$, then $M < N_{i+1}$. N_{i+1} is covered by some marking $M' \in \mathcal{M}$, so $M < N_{i+1} \leqslant M'$. Contradiction with $\mathcal{M}$ being minimal.

2) $M \notin \mathcal{R}$. M is the limit of the sequence of reachable markings (M_n). There is at least one element in $\mathcal{N}$ that covers all markings M_i, and it must have ω tokens in at least the same places as M. Let N be such an element, being the limit of the sequence (N_n). Applying the same logic, there must be an element $M' \in \mathcal{M}$ that covers all markings N_i, and $\Omega(M) \subseteq \Omega(M')$.

   Now, $\forall p \in \overline{\Omega}(M)$:

   - $N(p) \in \mathbb{N}$. Then $M(p) = M_i(p) \leqslant N(p) = N_i(p) \leqslant M'(p)$.
   - $N(p) = \omega$. Then $M(p) = M_i(p) \leqslant N_{M_i(p)} \leqslant M'(p)$.

   So $M \leqslant N \leqslant M'$, implying $M = N = M'$ by minimality of $\mathcal{M}$.

This goes for all elements in $\mathcal{M}$, and vice versa for the elements of $\mathcal{N}$, so $\mathcal{M} = \mathcal{N}$. □

As the MCS is unique, so is the coverability graph created from it, the *minimal coverability graph* MCG. The MCS and MCG are useful for answering several core problems regarding Petri nets:

- The Finite Reachability Tree Problem (FRTP) : The reachability tree is finite if and only if there are no ω tokens and no circuits.
- The Finite Reachability Set Problem (FRSP) : The reachability set is finite if and only if there are no ω tokens.
- The Boundedness Problem (BP) : Given a place p, is p bounded? This can be checked simply by looking at the elements of the MCS, p is unbounded if and only if it has an ω token in any marking of the MCS.
- The Quasi-Liveness Problem (QLP) : Given a transition t, can it fire during any execution of the net? This is checked during the construction of the MCG.
- The Coverability Problem (CP) : given a marking M, is there a reachable marking M' such that M is covered by M'? This is solved by checking whether M is covered by any element of the MCS. (For Petri nets, the Coverability Problem is reducible to the Quasi-Liveness Problem. In fact, a transition t is quasi-live if and only if there is a reachable marking M' such that for every place $p \in P$: $M'(p) \geqslant W(p,t)$, i.e. $M' \geqslant M = (W(p_1,t),...,W(p_{|P|},t))$. Conversely, the QLP is also reducible to the CP: a marking $M = (m_1.....m_p)$ is covered in a Petri net PN if and only if the associated transition t_M (defined as $\forall i \in \mathbb{N}_{\leqslant |P|} : W(p_i,t_M) = m_i$ and $W(t_M,p_i) = 0$) is quasi-live in the associated new Petri net "$PN + \{t_m\}$".)

The Regularity Problem (RP): given a Petri net PN, is *the language it describes regular*? The language of a petri net is the set of sequences enabled at the initial marking. This problem is a bit more complex, but "Petri nets and regular languages" [VVN81] covers it quite well: "THEOREM 3. A Petri net $N = (P,T,B,F,M_0)$ is regular, if and only if there is an integer k such that $\forall M \in R(N), \forall M' \in R(M), \forall p \in P : M'(p) \geqslant M(p) - k$.", where $R(N)$ is $\mathcal{R}$ and $R(M)$ is the set of markings reachable from M.

Example 4. Let Petri net PN so that there is only one place p, two transitions that add or substract one token from p, and $\hat{M} = \{0\}$. We can apply this theorem: Let $M = \{k+1\} \in R(N), M' = \{0\} \in R(M)$, but $0 < k+1-k \rightarrow$ the language PN describes is not regular.

In other words, the language L(PN) is regular if and only if every elementary circuit of MCG(PN) containing an unbounded marking is labelled by a sequence of transitions $x \in T^+$ such that $D(p,x) \geqslant 0$ for every place p. As it is trivial to generate the coverability graph when a coverability set is known, all algorithms that can calculate the MCS solve this problem as well.

2.4 Calculating Coverability Sets

If $\mathcal{R}$ is finite, the MCS is simply the set of maximal elements of $\mathcal{R}$.

Theorem 2.3. The set of reachable markings of a Petri net PN is infinite if and only if the pumping lemma is usable on some pair of reachable markings.

$\Leftarrow$: Let the pumping lemma be usable on (M, M'), where $\hat{M}\,[\,\sigma\,\rangle\,M$ and $M\,[\,\tau\,\rangle\,M'$. The infinite amount of M_n are all reachable: $\forall n \in \mathbb{N} : \hat{M}\,[\,\sigma\tau^n\,\rangle\,M_n$. □

$\Rightarrow$: Consider the reachability tree of PN. We apply Königs lemma: As the set of reachable markings is infinite, and the degree of branching at any node is at most $|T|$, there must be an infinite path, with an infinite amount of distinct markings. Construct an infinite sequence from this path by discarding all occurrences of markings beyond their first occurrence. We apply lemma 4: there is an infinite strictly increasing sequence, there are nodes M, M' in this path, $\exists \sigma \in T^+ : M\,[\,\sigma\,\rangle\,M'$, and $M < M'$, so the pumping lemma is usable.

With ω-markings, we can generate a "reachability tree"-like graph, a *Karp&Miller tree* [KM69]. While exploring the set of reachable markings, similar to the reachability tree, we use the pumping lemma. When two nodes n, n' are found, where n is an ancestor of n' and $M = \Lambda(n) < M' = \Lambda(n')$, we replace the labelling $\Lambda(n') := M''$ where:

$$\forall p \in P : M''(p) = \begin{cases} M'(p) & \text{if } M(p) = M'(p) \\ \omega & \text{otherwise} \end{cases}$$

If we find a node labelled identically to one of its ancestors, we don't explore that node. This ensures that this algorithm terminates [KM69], and will produce a Karp&Miller tree. Figure 2.7 shows a Karp&Miller tree of the net in Figure 2.4.

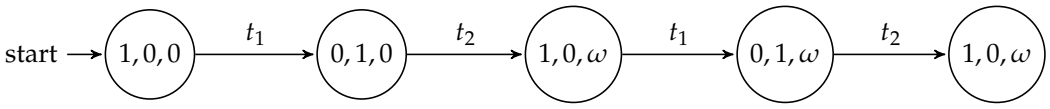


Figure 2.7: A Karp&Miller tree of Figure 2.4.

While it is finite, the Karp&Miller tree is by no means efficient: it grows exponentially with the amount of parallelism possible in the net. The net in Figure 2.6 has an MCS of just four elements, but the K&M tree has 168 nodes, with 16 unique markings. There are many nodes with identical markings, and their subtrees are identical too, if no further pumping occurs. All the explored markings are kept in memory, making the K&M tree impractical in most cases. As the K&M algorithm is from 1969, better algorithms have been found for calculating the minimal coverability set. In this thesis, we will compare several algorithms that can calculate the MCS.

2.5 History of Coverability Sets

It was C.A. Petri's intent from the start to model distributed asynchronized computer systems with Petri nets [BR09]. A modular approach turned out to be the best approach for many current day solutions, both software and hardware: Data centres, search engines, and crowd computing all utilize a modular approach.

The usage of Petri nets to model physical objects did not need unboundedness, as there are finite resources available. When Petri nets were used as workflow management tools in electronic systems, the usefulness of coverability sets became apparent. Especially if the workflow is that of independent computer modules, such as a distributed network service, it's useful to observe unwanted behaviour, like deadlock or non-terminating programs. This can be detected by the fact that a coverability set contains ω-markings.

Some problems are harder, like the *reachability problem* : Given a marking M, is it reachable? An ω token is no guarantee, for it could mean that the place is unbounded for values larger than the target value, or that only odd values are possible while the target value is even. After more than ten years, it has been proven that the reachability problem is decidable. A proof is explained in e.g. [Reu90].

It has been proven that "the coverability and reachability problems are undecidable for generalized Petri nets in which a distinguished transition had priority over the other transitions" [AK76]. Indeed, if two places p_1 and p_2 are simultaneously unbounded, but no transition that increments p_2 has priority, you will not find p_2 to grow when exploring the reachability tree. But ω-markings give us the functionality we need to solve this problem.

Another unsolved problem was the Reachability Set Inclusion Problem of Vector Addition Systems, i.e. whether the reachability set of a Petri net is a subset of the reachability set of another net. This has been proven to be indecidable [Bak73].

The MCS and ω-markings have proven to be imperative in analysing Petri nets. Other authors have found efficient ways to calculate the MCS, without constructing the entire K&M tree, or by avoiding the tree altogether. The algorithms investigated try to improve in either speed or memory usage using new insights.

Chapter 3

The algorithms

We compare five papers detailing algorithms to calculate the MCS. All "quoted text" in the sections are quotes to the paper of that section.

3.1 Parallel program schemata

by Richard M. Karp and Raymond E. Miller [KM69]

This invaluable paper is not strictly about Petri nets, but about Parallel Program Schemata. This is a larger group of models, as for any operation, the initiation and the termination are two distinct events, allowing more strict modelling of parallelism.

> A program can be regarded as a collection of primitive operations which depend on and affect memory locations, together with a specification of the rules governing the initiation and termination of operations.
>
> Definition 1.1. A parallel program schema $\Phi = (M, A, T)$ is specified by:
>
> 1. A set M of *memory locations*,
> 2. A finite set $A = \{a, b, \ldots\}$ of *operations* and, for each operation a in A:
>
>   (a) a positive integer $K(a)$ called the *number of outcomes* of a;
>
>   (b) a set $D(a) \subseteq M$ whose elements are the *domain locations* for a;
>
>   (c) a set $R(a) \subseteq M$ whose elements are the *range locations* for a.
> 3. A quadruple $\mathcal{T} = (Q, q_0, \Sigma, \tau)$ called the *control*, where:
>
>   (a) Q is a set of *states*;
>
>   (b) q_0 is a designated state called the *initial state*;

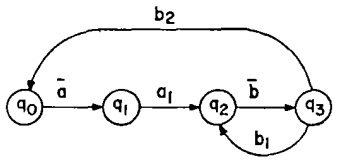


Figure 3.1: The control T for a finite-state schema Φ. [KM69]

(c) Σ, the *alphabet*, is the union of

$$\sum_i = \bigcup_{a \in A} \{\overline{a}\}$$

the *initiation symbols* and

$$\sum_t = \bigcup_{a \in A} \{a_1, \ldots, a_{K(a)}\}$$

the *termination symbols*.

(d) τ, the *transition function*, is a partial function from $Q \times \Sigma$ into Q which is total on $Q \times \Sigma_t$.

Definition 1.2. An *interpretation* ϕ of a schema Φ is specified by:

1. a function C associating with each element $i \in M$ a set $C(i)$;
2. an element $c_0 \in \times_{i \in M} C(i)$;
3. for each operation a, two functions:

$$F_a : \bigtimes_{i \in D(a)} C(i) \to \bigtimes_{i \in R(a)} C(i)$$

$$G_a : \bigtimes_{i \in D(a)} C(i) \to \{a_1, a_2, \ldots, a_{K(a)}\}$$

Page 149-151, [KM69].

"Elements of Σ_i denote initiations of operations and elements of Σ_t denote terminations of operations with given outcomes." Consider the control function in figure 3.1. a might set a counter to 0, while b increments this counter by 1, and returns whether this value is smaller than 10(where b_1 means true and b_2 means false). But b might also never return true, or never return false! Merely a schema without an interpretation is not that useful. It is for this reason that they extend this definition to *counter schemata*, which have the characteristics of Vector Addition Systems in them.

"Because vector addition systems underly all of our decision procedures, we begin by discussing these systems as mathematical structures in their own right." (page 165-169) Their definition of a reachability set $R(\mathcal{W})$ of a vector addition system $\mathcal{W}$ is the set of all vectors of the form $d + w_1 + w_2 + \cdots + w_s$ such that $w_i \in W : i = 1, 2, \ldots, s$ and $d + w_1 + w_2 + \cdots + w_i \geqslant 0 : i = 1, 2, \ldots, s$. Petri nets are vector addition systems as well, and this matches our definition of the reachable set for Petri nets. They then present what is later

known as a Karp&Miller tree, where they associate a rooted tree $\mathcal{T}(\mathcal{W})$ with a vector addition system $\mathcal{W}$. Note that no algorithm is given to construct this tree, it is simply described in rules:

- A *rooted tree* is a directed graph such that one vertex (the *root* δ) has no edges directed into it,
- each other vertex has exactly one edge directed into it,
- and each vertex is reachable from the root.
- If ζ and η are distinct vertices of a rooted tree, and there is a path from ζ to η, then we say $\zeta \prec \eta$;
- if there is an edge from ζ to η, then η is a *successor* of ζ.
- A vertex without successors is called an *end*.
- A *labelling* $l(\zeta)$ assigns to each vertex ζ an r-dimensional vector whose coordinates are elements of $\mathbb{N} \cup \{\omega\}$.

(1) The root is labelled d;

(2) Let η be a vertex;

   (a) if, for some vertex $\zeta \prec \eta$, $l(\zeta) = l(\eta)$, then η is an end.

   (b) otherwise, the successors of η are in one-to-one correspondence with the elements $w \in \mathcal{W}$ such that $0 \leqslant l(\eta) + w$. Let the successor of η corresponding to w be denoted by η_w. For each i the ith coordinate of the label $l(\eta_w)$ is determined as follows:

      (i) if there exists $\zeta \prec \eta_w$ such that $l(\zeta) \leqslant l(\eta) + w$ and $l(\zeta)_i < (l(\eta) + w)_i$ then $l(\eta_w)_i = \omega$;

      (ii) if no such ζ exists, then $l(\eta_w)_i = (l(\eta) + w)_i$.

"We remark that, since $\mathcal{T}(\mathcal{W})$ is finite, its construction, using the recursive definition, is clearly effective."

It is then proven that this tree is always unique, finite, and can be used to solve a number of problems, like the coverability problem, and whether coordinates can be simultaneously unbounded. It's also shown that Vector Addition Systems can be transformed into counter schemata without loss of behaviour, and vice versa: Let $(P, T, W, \hat{M})$ be a Petri net. The memory locations are the places P, the operations a are the transitions T, and they only have one outcome, success. Let Q contain the reachable set, and let q_0 be the initial marking $\hat{M}$. Then $M\,[\,a\,\rangle\,M'$ if and only if

- τ maps from $(M, \overline{a})$ to $\overline{M}$,
- τ maps from $(\overline{M}, a_1)$ to M',
- τ maps no other pair $(\overline{M}, \overline{a'})$,
- $D(a) \subseteq R(a)$,
- F_a describes $W(a, i)$ and $W(i, a)$,
- G_a maps to success in all cases.

The rest of the paper focuses on schemata and is outside of the scope of this thesis.

3.2 The minimal coverability graph for Petri nets

by Alain Finkel [Fin91]

"We arc interested in algorithms which take a Petri net PN and a property π in input and answer automatically, after a finite delay, whether or not PN satisfies π."

The properties in question are those that can be decided by the Karp&Miller tree: "the Boundedness Problem (BP), the Finite Reachability Tree Problem (FRTP), the Finite Reachability Set Problem (FRSP), the Quasi-Liveness Problem (QLP) or the equivalent problem called the Coverability Problem (CP) and the Regularity Problem (RP). . . . One of our aims here is to define a graph which permits to decide upon these five problems while, at the same time, being faster to compute and taking less space than the Karp-Miller graph."

We remark that their definition of a coverability set is the same as ours, as it "is a set CS of markings such that: 1) it covers all the markings of the reachability set and 2) for each marking m' in CS but not in the reachability set, there is an infinite strictly increasing sequence of reachable markings $\{m_n\}$ converging to m'", where *converging* is equivalent to our definition of a limit.

The paper then explains the minimal_coverability_graph(MCG) and minimal_coverability_tree(MCT) procedures. "There are four basic ideas for constructing the *minimal coverability graph*", which has the same definition as our MCG.

- The first idea is to develop the Karp-Miller tree until we meet two markings m and m' such that $m \geqslant m'$, instead of $m = m'$.

- The second idea is to compact the previous reduced Karp-Miller tree during its development. We continue a marking m' if and only if m' is incomparable with any computed markings. If $m < m'$, s.t. there is a path from m to m', we compute a new marking m'' such that for every marking m, we have: for every place p, if $m'(p) > m(p)$ then $m''(p) := \omega$ else $m''(p) := m'(p)$. Let n be the highest node m used, we change its label to m'' and we remove its subtree.

- The third idea consists of removing every subtree whose root is labelled by a marking m such that $m < m''$. The idea is this will not produce new markings, but this is later proven faulty.

- Finally, the fourth idea is first to identify two nodes which have the same label; and mostly to only keep arcs (m, t, m') such that the transition t is fireable from m and we reach the marking m' exactly, i.e. we discard the pumping transitions.

The MCT procedure uses these first three ideas to make a non-unique *minimal coverability tree*. The MCG first removes arcs that are no longer representations of a transition firing to get the unique *minimal coverability*

forest, and then identifies nodes with each other to get the unique *minimal coverability graph*. The calculations made by the MCT procedure seem solid at first. However, a decade later, it is proven wrong. A counterexample to the algorithm is shown in Figure 3.5.

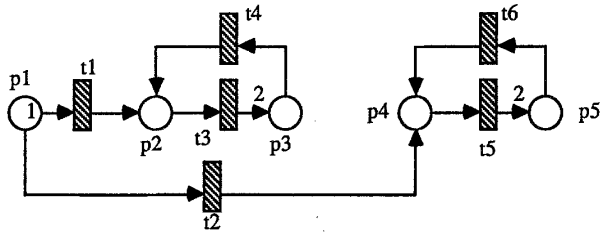


Figure 2.1: Petri net PN1.

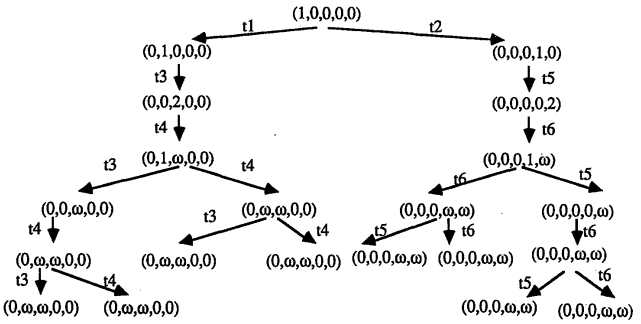


Figure 2.3: the Karp-Miller tree of PN1 (21nodes, 20 arcs)

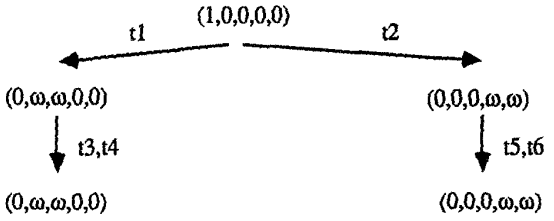


Figure 4.1.(d). The final MCT. We stop because m=m' with m-*->m'.

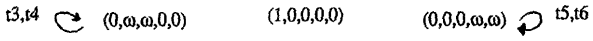


Figure 4.1.(e). MCG. We have identified nodes having the same label and removed useless arcs.

Figure 3.2: Original figures detailing the effect of MCT and MCG. [Fin91]

" **Definition 7.1.** A **coverability tree** CT(PN) of a Petri net $PN = \{P, T, W, \hat{M}\}$ is a labelled directed tree $\{N, L, A\}$ where the set of nodes N is a coverability set of PN, $L = T$ and arcs (n, t, n') of A (with label$(n) = m$ and label$(n') = m'$) are of types (1) or (2) :

(1) if $m\,[\,t\rangle\, m'$ then there is an arc $(n, t, n') \in A$;

(2) if $m\,[\,t\rangle\, \cdot$ and if (not ($m\,[\,t\rangle\, m'$)) then there is an infinite sequence of finite sequences of transitions $\{tx_n\}$ such that for every $n \geqslant 0$, the sequence of transitions tx_n is fireable from m, we reach a marking m_n and the sequence $\{m_n\}$ converges to m': $m\,[\,tx_n\rangle\, m_n$and $\lim m_n = m'$.

Definition 7.2. A **minimal coverability tree** MCT(PN) of a Petri net PN, is a coverability tree such that its set of nodes is the minimal coverability set. The *unique minimal coverability forest* of a Petri net, MCF(PN), is obtained from a minimal coverability tree by removing all arcs of type (2).

Proposition 7.3.

1) There is not a unique minimal coverability tree.

2) There is a unique minimal coverability forest.

3) The minimal_coverability_tree procedure computes a minimal coverability tree." [Fin91]

The five problems (FRTP, FRSP, BP, QLP, RP) are all decidable with the minimal coverability graph (Corollary 3.17 [Fin91]).

Theorem 3.1. Let PN be a Petri net, MCG(PN) its minimal coverability graph and MCS(PN) its minimal coverability set.

1) The reachability tree RT(PN) is infinite if and only if there is at least one circuit in MCG(PN).
2) The reachability set RS(PN) is infinite if and only if there is at least one symbol ω in MCG(PN).
3) A place p is not bounded if and only if there is at least one marking $M \in$ MCS(PN) such that $M(p) = \omega$
4) A transition t is quasi-live if and only if there is at least one marking $M \in$ MCS(PN) such that for every place p, $M(p) > V(p,t)$.
5) The language L(PN) is regular if and only if every elementary circuit of MCG(PN) containing an infinite marking is labelled by a sequence of transitions $x \in T^+$ such that $D(p,x) \geqslant 0$ for every place p.

For completeness, here is the algorithm description from the paper.

procedure minimal_coverability_graph(PN: Petri net; var MCS: set of markings; var MCG: graph);
{* the result will be in MCG *}
begin
minimal_coverability_tree(PN; MCS; MCT);
identify_nodes_having_same_label(MCT; MCG);
{* the procedure "identify_nodes_having_same_label(T: tree; G: graph)" transforms the tree T into a graph G such that two nodes in T having the same label are identified in G *}
for every arc (m,t,m') of MCG **do**
{* after having identify nodes with the same label, we confuse without ambiguity a node and its label *}
if not ($m(t > m'$) **then** remove_arc((m,t,m'); MCG);
{* the procedure "remove_arc((m,t,m'); var G))" only removes the arc (m,t,m') from G *}
end;

Figure 3.3: Original description of the minimal_coverability_graph procedure.

```
procedure minimal_coverability_tree(PN: Petri net; var MCS: set of markings; var MCT: tree);
{* the result will be in MCT *}
var unprocessednodes, processednodes: set of nodes; n, n', n_l, n_2: node; t: transition; ancestor: boolean;
    begin
        unprocessednodes := { create_node(r, M_0) };                    {* M_0 is the marking of root r *}
        processednodes := ∅;                                            {* processednodes will be the minimal coverability set *}
        while unprocessednodes ≠ ∅ do
            begin
                select some node n ∈ unprocessednodes;
                unprocessednodes := unprocessednodes - {n};
                case n : [1..4] of                                      {* m is the marking of n and m_1 is the marking of n_1 *}
                    1: there is a node n_1 ∈ processednodes such that m = m_1 :
                        begin
                            processednodes := processednodes + {n};
                            exit;                                       {* exit of case *}
                        end;
                    2: there is a node n_l ∈ processednodes such that m < m_1 :
                        begin
                            remove_node(n; MCT);
                            exit;                                       {* exit of case *}
                        end;
                        {* the procedure "remove_node(n: node; var T: tree)" removes the node n and the arc from the
                        direct ancestor of n to n, in the tree T *}
                    3: there is a node n_1 ∈ processednodes such that m_1 < m :
                        begin
                            m_2 := m; ancestor := false;
                            for all ancestors n_1 of n such that m_1 < m do
                                    for all places p such that m_1(p) < m(p) do m_2(p) := ω;
                            if there is an ancestor n_1 of n such that m_1 < m_2 then
                                begin
                                    ancestor := true;
                                    n_1 := first node processed, on the path from the root to n such that m_1 < m_2;
                                    m_1 := m_2;
                                    remove_tree(n_1; MCT);
                                    { * the procedure "remove_tree(n: node; var T: tree)" which removes the subtree
                                    whose root is n in the tree T (note that we keep the root n) * }
                                    remove from(processednodes+unprocessednodes) all nodes of tree(n_1;MCT);
                                    unprocessednodes := unprocessednodes + {(n_1};
                                end;
                            for every n_1 ∈ processednodes such that m_1 < m_2 do
                                begin
                                    remove from(processednodes+unprocessednodes) all nodes of tree(n_1;MCT);
                                    remove_tree(n_1; MCT);
                                    remove_node(n_1; MCT);
                                end;
                            if ancestor = false then unprocessednodes := unprocessednodes + {n};
                            exit;                                       {* exit of case *}
                        end;
                    4: otherwise :
                        begin
                            for every transition t such that m(t > m' do
                                begin
                                    create_node+arc((n, t, n'); MCT);   { * m' is the marking of the new node n' * }
                                    { * the procedure "create_node+arc((n, t, n'); T)" creates a new node n' labelled
                                    by m' and a new arc (n, t, n') in the tree T * }
                                    unprocessednodes := unprocessednodes + {n'};
                                end;
                            processednodes := processednodes + {n};
                            exit;                                       { * exit of case * }
                        end;
                end;                                                    {* end of case *}
                unprocessednodes := maximal(unprocessednodes);
                { * the function "maximal(S : set) : set" computes the set of nodes n such that every label(n) is maximal * }
                MCS := { label(n) ; n ∈ processednodes };
            end;;                                                       {* end of while *}
    end;;                                                               {* end of procedure *}
```

Figure 3.4: Original description of the minimal_coverability_tree procedure.

3.3 On the efficient computation of the minimal coverability set of Petri nets

by Gilles Geeraerts, Jean-Francóis Raskin and Laurent Van Begin [GRB10]

This is one of the papers that proves the MCT algorithm to be wrong, as mentioned earlier. It gives us a specific counterexample Petri net, and shows that the execution of MCT does not return the MCS. "The flaw is intricate and we do not see an easy way to get rid of it." The original images used to explain the counterexample can be found in Figure 3.5. Instead of fixing this, they then show a new algorithm for computing the MCS, focused on being correct rather than sacrificing correctness for speed. The experimental results show that it behaves much better in practice than the KM algorithm.

"It is based on novel ideas: first, we do not build a tree but handle sets of pairs of markings. Second, in order to exploit the monotonicity property, we define an adequate order on pairs of markings that allows us to maintain sets of maximal pairs only. As a consequence, our new solution manipulates sets that are minimal too (as the MCT intends to do), but for a different ordering (an ordering on pairs, instead of an ordering on markings)."

"Given an ω-marking m of some PN $\mathcal{N} = \langle P, T \rangle$, we let $\text{Post}(m) = \{m' | \exists t \in T : m [t \rangle m'\}$ and $\text{Post}(m) = \{m' | \exists \sigma \in T : m [\sigma \rangle m'\}$." The reachable set is thus $\text{Post}(\hat{M})$.

"Given a set M of ω-markings, we define the set of *maximal elements of* M as $\max^{\leqslant}(M) = \{m \in M | \nexists m' \in M : m < m'\}$. Given an ω-marking m (ranging over set of places P), its *downward-closure* is the set of markings $\downarrow^{\leqslant}(m) = \{m' \in \mathbb{N}^{|P|} | m' \leqslant m\}$. Given a set M of ω-markings, we let $\downarrow^{\leqslant}(M) = \bigcup_{m \in M} \downarrow^{\leqslant}(m)$. A set $D \subseteq \mathbb{N}^{|P|}$ is *downward-closed* iff $\downarrow^{\leqslant}(D) = D$." Every set covers its downward closure, and the downward closure of an ω-marking M is an infinite set that grows arbitrarily large in places $\Omega(M)$.

"**Definition 4.** Let $\mathcal{N} = \langle P, T \rangle$ be a PN and let m_0 be the initial ω-marking of $\mathcal{N}$. The *covering set* of $\mathcal{N}$, denoted as $\text{Cover}(\mathcal{N}, m_0)$ is the set $\downarrow^{\leqslant}(\text{Post}^*(m_0))$. ...a coverability set for $\mathcal{N}$ and m_0 is a finite sub-set $S \subseteq (\mathbb{N} \cup \{\omega\})^{|P|}$ such that $\downarrow^{\leqslant}(S) = \text{Cover}(\mathcal{N}, m_0)$." The Cover is the downward closure of the reachability set, equal to the downward closure of any coverability set.

"Given a set R of pairs of ω-markings, we let $\text{Flatten}(R) = \{m | \exists m' : (m', m) \in R\}$. We use the $\overline{\text{Post}}$ function to define the notion of successor of a pair of ω-markings (m1,m2): $\overline{\text{Post}}((m_1, m_2)) = \{(m_1, m'), (m_2, m') | m' \in \text{Post}(m_2)\}$." A pair (m_1, m_2) basically means that m_2 is reachable from m_1. Any successor of m_2 is reachable from both m_1 and m_2.

The acceleration in the K&M tree is called the function $\text{Accel}(S, m)$, where S are the ancestors of m, and the return value is the new marking with added ω tokens. "Our new solution relies on

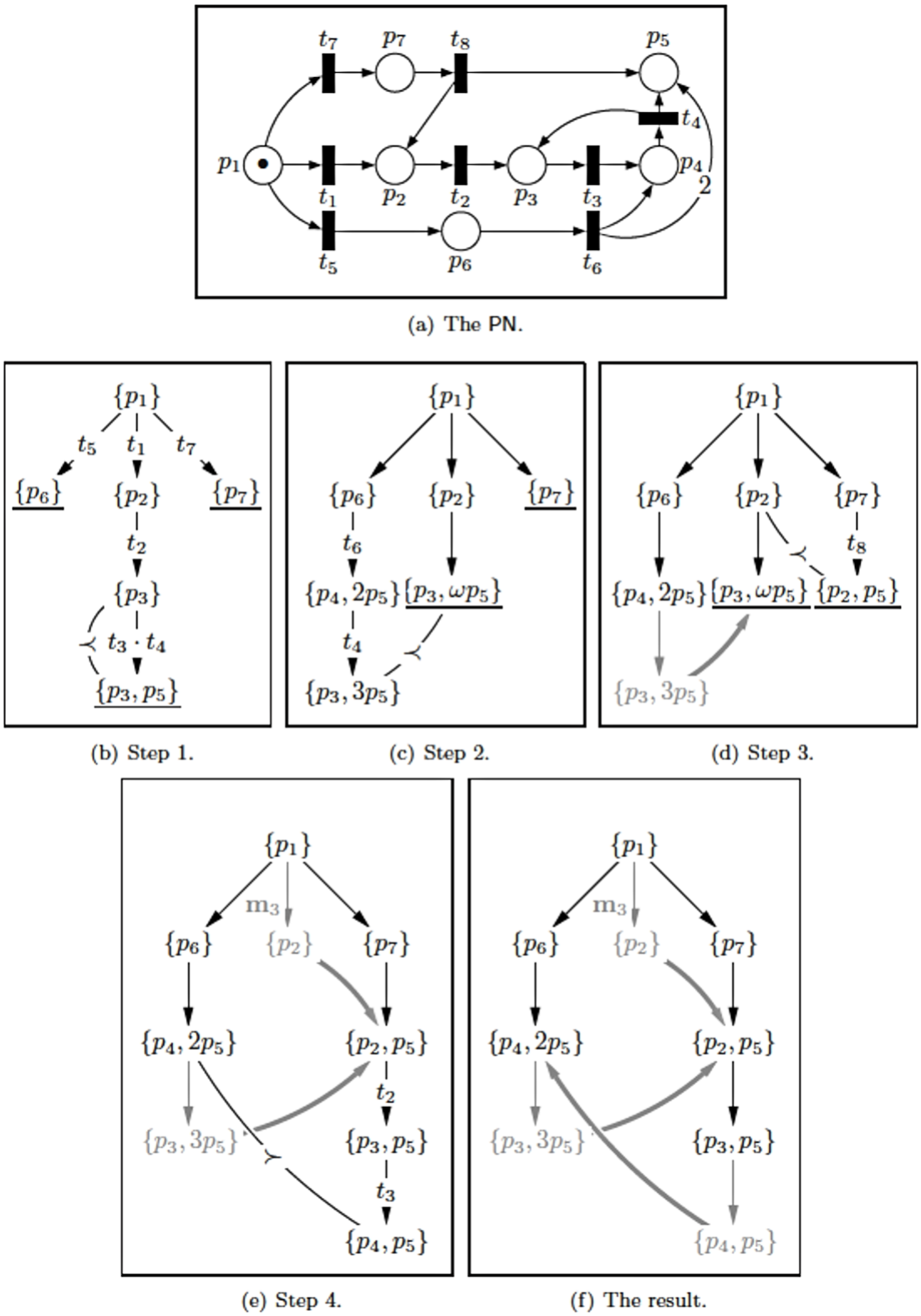


Figure 3.5: Original figure with counterexample to MCT algorithm. [GRB10]
A counter-example to the MCT algorithm. Underlined markings are in the frontier. A gray arrow from n to n' means that n' is the reason n was deactivated. Note how the execution returns an incorrect set. In step 3, the marking $\{p_3, \omega p_5\}$ is deactivated, though no marking covering it is generated from $\{p_2, p_5\}$.

a weaker acceleration function than that of Karp&Miller (because its first argument is restricted to a single marking instead of a set of markings). Given two ω-markings m_1 and m_2 s.t. $m_1 \leqslant m_2$, we let AccelPair$(m1, m2) = m_\omega$ s.t. for any place $p, m_\omega(p) = m_1(p)$ if $m_1(p) = m_2(p); m_\omega(p) = \omega$ otherwise. Similarly to Post, we use $\overline{\text{Accel}}$ to define the notion of acceleration of a pair of ω-markings (m_1, m_2) : $\overline{\text{Accel}}((m_1, m_2)) = \{(m2, \text{AccelPair}(m_1, m_2))\}$ if $m_1 \prec m_2$; and $\overline{\text{Accel}}((m_1, m_2))$ is undefined otherwise. We extend the $\overline{\text{Post}}$ and $\overline{\text{Accel}}$ functions to sets R of pairs in the following way: $\overline{\text{Post}}(R) = \bigcup_{(m_1,m_2)\in R} \overline{\text{Post}}((m_1, m_2))$ and $\overline{\text{Accel}}(R) = \bigcup_{(m_1,m_2)\in R, m_1 \prec m_2} \{\overline{\text{Accel}}(m_1, m_2)\}$."

"Now that we have defined function to handle the semantics of the PN in terms of pairs, we define the ordering $\sqsubseteq$ that will allow us to reduce the size of the sets our algorithm manipulate. Let m_1 and m_2 be two ω-markings. Then, $m_1 \ominus m_2$ is a function $P \mapsto \mathbb{Z} \cup \{-\omega, \omega\}$ s.t. for any place $p : (m_1 \ominus m_2)(p)$ is equal to ω if $m_1(p) = \omega$; $-\omega$ if $m_2(p) = \omega$ and $m_1(p) \neq \omega; m_1(p) - m_2(p)$ otherwise. Then, given two pairs of ω-markings (m_1, m_2) and (m'_1, m'_2), we have $(m_1, m_2) \sqsubseteq (m'_1, m'_2)$ iff $m_1 \leqslant m'_1, m_2 \leqslant m'_2$ and for any place $p : (m_2 \ominus m_1)(p) \leqslant (m'_2 \ominus m'_1)(p)$." In other words, pair$_1 \sqsubseteq$ pair$_2$ if pair$_1$ is obsolete: pair$_1$ has smaller markings *with less growth*. Remark that $\ominus$ never results in $-\omega$, as $\langle m_1, m_2\rangle$ means that m_2 is reachable from m_1, and thus $\Omega(m_1) \subseteq \Omega(m_2)$.

"Example 11. *Let us assume a PN with two places. Then:* $(\langle 0,1\rangle, \langle 0,2\rangle) \sqsubseteq (\langle 1,1\rangle, \langle 2,5\rangle)$ *and* $(\langle 0,1\rangle, \langle 0,2\rangle) \sqsubseteq (\langle 1,\omega\rangle, \langle 2,\omega\rangle)$. *However,* $(\langle 0,1\rangle, \langle 0,2\rangle) \not\sqsubseteq (\langle 1,7\rangle, \langle 2,7\rangle)$ *although* $\langle 0,1\rangle \leqslant \langle 1,7\rangle$ *and* $\langle 0,2\rangle \leqslant \langle 2,7\rangle$. *Indeed,* $\langle 0,2\rangle \ominus \langle 0,1\rangle = \langle 0,1\rangle, \langle 2,7\rangle \ominus \langle 1,7\rangle - \langle 1,0\rangle$ *and* $\langle 0,1\rangle \not\leqslant \langle 1,0\rangle$."

Analogous to the $\downarrow^{\leqslant}$and Max$^{\leqslant}$ on markings, we define $\downarrow^{\sqsubseteq}$and Max$^{\sqsubseteq}$ on pairs: "For any (m_1, m_2), we let $\downarrow^{\sqsubseteq}((m_1, m_2)) = \{(m'_1, m'_2) | (m'_1, m'_2) \sqsubseteq (m_1, m_2)\}$. We extend this to sets of pairs R as follows: $\downarrow^{\sqsubseteq}(R) = \bigcup_{(m_1,m_2)\in R} \downarrow^{\sqsubseteq}((m_1, m_2))$. Given a set R of pairs of markings, we let $\text{Max}^{\sqsubseteq}(R) = \{(m_1, m_2) \in R | \nexists (m'_1, m'_2) \in R : (m_1, m_2) \neq (m'_1, m'_2) \wedge (m_1, m_2) \sqsubseteq (m'_1, m'_2)\}$.

"**Definition 14.** Given a Petri net $\mathcal{N}$ and an initial ω-marking m_0, an *oracle* is a function Oracle : $\mathbb{N} \mapsto (\mathbb{N} \cup \{\omega\})^{|P|} \times (\mathbb{N} \cup \{\omega\})^{|P|}$ that returns, for any $i \geqslant 0$, a set of pairs of ω-markings that satisfies the two following conditions:

$$\downarrow^{\leqslant}(\text{Post}(\text{Flatten}(\text{Oracle}(i)))) \subseteq \downarrow^{\leqslant}(\text{Flatten}(\text{Oracle}(i)))$$
$$\downarrow^{\leqslant}(\text{Flatten}(\text{Oracle}(i))) \subseteq \text{Cover}(\mathcal{N}, m_0).$$"

In other words, each set Flatten(Oracle(i)) is closed under Post, and does not contain any unreachable markings.

Now that we're up to speed on the definitions this paper uses, we can get to the definition of their procedure for calculating the MCS. "**Definition 15.** Let $\mathcal{N} = \langle P, T\rangle$ be a PN, m_0 be an initial marking, and Oracle be an oracle. Then, the *covering sequence of* $\mathcal{N}$, noted CovSeq$(\mathcal{N}, m_0, \text{Oracle})$ is the infinite sequence $(V_i, F_i, O_i)_{i\geqslant 0}$, defined as follows:

- $V_0 = \varnothing, O_0 = \varnothing$ and $F_0 = \{(m_0, m_0)\}$
- For any $i \geqslant 1 : O_i = \text{Max}^{\sqsubseteq}(O_{i-1} \cup \text{Oracle}\ (i))$;
- For any $i \geqslant 1 : V_i = \text{Max}^{\sqsubseteq}(V_{i-1} \cup F_{i-1}) \setminus \downarrow^{\sqsubseteq}(O_i)$;
- For any $i \geqslant 1 : F_i = \text{Max}^{\sqsubseteq}(\overline{\text{Post}}(F_{i-1}) \cup \overline{\text{Accel}}(F_{i-1})) \setminus \downarrow^{\sqsubseteq}(V_i \cup O_i)$.

A covering sequence is thus a sequence of triples (V_i, F_i, O_i), for any $i \geqslant 0$, where all the V_i, F_i and O_i are sets of pairs of markings. O_i represents all the information provided by the *oracle*, V_i are the *visited* pairs, that were once in the *frontier* F_i. "Remark that, in all these sets, only maximal (wrt to $\sqsubseteq$) pairs are kept. This allows to keep the size of these sets smaller." They then continue to prove that

> **Theorem 27.** *Let $\mathcal{N}$ be a PN, m_0 be its initial marking,* Oracle *be an oracle, and* $\text{CovSeq}(\mathcal{N}, m_0, \text{Oracle}) = (V_i, F_i, O_i)_{i \geqslant 0}$. *Then, there exists $k \geqslant 0$ such that*
> *(1) for all $1 \leqslant i < k$:$\downarrow^{\leqslant}(\text{Flatten}\ (V_i \cup O_i)) \subset \downarrow^{\leqslant}(\text{Flatten}\ (V_{i+1} \cup O_{i+1}))$;*
> *(2) for all $i \geqslant k$:$\downarrow^{\leqslant}(\text{Flatten}\ (V_i \cup O_i)) = \text{Cover}\ (\mathcal{N}, m_0)$.*

In other words, after some amount of steps k the sequence stabilizes, and $V_i \cup O_i$ is the MCS. Building the covering sequence in a bounded Petri net is comparable to building the K&M tree in a breadth-first-search manner, as $\text{Flatten}(F_i)$ is equal to the set of nodes of the K&M tree at depth i.

This works for any Oracle. The empty Oracle, for which $\text{Oracle}(i) = \varnothing$ for all $i \geqslant 1$, has correct results. The Oracle can be used to feed information we have already gathered to the algorithm. Some subset of the MCS, that is closed under Post, might already be known. This is used in the **CoverProc**$(\mathcal{N}, m_0)$ algorithm, which calculates the MCS of a Petri net $\mathcal{N}$ with *initial marking* m_0. The oracle starts out empty. Whenever an acceleration from m to m' is encountered, the procedure recursively calls itself with $\text{CovProc}(\mathcal{N}, m')$. The recursive call calculates the subset of the MCS that has ω tokens in at least locations $\Omega(m')$. *At least*, because there are recursive calls every acceleration. Every call layer continues until it is stable, i.e.: the set $\downarrow^{\leqslant}(\text{Flatten}\ (O_i \cup V_i)) \subseteq \downarrow^{\leqslant}(\text{Flatten}\ (O_{i-1} \cup V_{i-1}))$. The resulting $O_i \cup V_i$ are then passed on to the previous layer, and used as an Oracle.

This approach mixes breadth-first-searching for an acceleration with depth-first exploring when an acceleration is found. In only the deepest layers of the recursive call, the algorithm continues until a closed set of *visited* pairs is found, and all layers above it quickly find that almost all their pairs are smaller (wrt $\sqsubseteq$) than the Oracle, stabilizing their own sequence.

The algorithm is shown in Figure 3.6, and an execution on the counterexample net is shown in Figure 3.7. After iteration 6, the algorithm finds $\downarrow^{\leqslant}\text{Flatten}(O_5 \cup V_5) = \downarrow^{\leqslant}\text{Flatten}(O_6 \cup V_6)$ and returns the MCS.

At the end of the paper are some results, comparing a K&M algorithm, a covering sequence with an empty oracle, and CoverProc. It states that CovProc has a maximum of 4 pairs at a time during the *kanban* net, pictured in *B*.4. Kanban starts out with three ω tokens, and the MCS is $\{\omega^{16}\}$, so there are 13 accelerations.

Data: A PN $\mathcal{N} = \langle P, T \rangle$, an initial ω-marking $\mathbf{m}_0$
Result: A set of pairs of markings.
$\mathsf{CovProc}\,(\mathcal{N}, \mathbf{m}_0)$ **begin**
  $i := 0$; $\overline{O}_0 := \emptyset$; $\overline{V}_0 := \emptyset$; $\overline{F}_0 := \{(\mathbf{m}_0, \mathbf{m}_0)\}$;
  repeat
    $i := i + 1$;
    $R_i := \cup_{\mathbf{m} \in S} \mathsf{CovProc}\,(\mathcal{N}, \mathbf{m})$ where $S \subseteq \mathsf{Flatten}\,(\overline{\mathsf{Accel}}\,(F_{i-1}))$;
    $\overline{O}_i := \mathsf{Max}^{\sqsubseteq}\,(\overline{O}_{i-1} \cup R_i)$;
    $\overline{V}_i := \mathsf{Max}^{\sqsubseteq}\,(\overline{V}_{i-1} \cup \overline{F}_{i-1}) \setminus {\downarrow}^{\sqsubseteq}(\overline{O}_i)$;
    $\overline{F}_i := \mathsf{Max}^{\sqsubseteq}\,(\overline{\mathsf{Post}}\,(\overline{F}_{i-1}) \cup \overline{\mathsf{Accel}}\,(\overline{F}_{i-1})) \setminus {\downarrow}^{\sqsubseteq}(\overline{O}_i \cup \overline{V}_i)$;
  until ${\downarrow}^{\preccurlyeq}(\mathsf{Flatten}\,(\overline{O}_i \cup \overline{V}_i)) \subseteq {\downarrow}^{\preccurlyeq}(\mathsf{Flatten}\,(\overline{O}_{i-1} \cup \overline{V}_{i-1}))$;
  `return`$(\overline{O}_i \cup \overline{V}_i)$;
end

Figure 3.6: The CovProc algorithm [GRB10]

The maximum amount of pairs is calculated as $\max\{|V_i \cup O_i \cup F_i|, i \geqslant 1\}$, but it only accounts for that layer. The recursive calls of CovProc don't count each other's pairs. The K&M execution of kanban has way more than ten thousand nodes, the execution stopped at 1222 seconds, more than the designated timeout period of two minutes; filling the table was done inconsistently. Creating 16 nodes with a K&M algorithm takes more time than generating 47 pairs(and thus 94 markings) in the CoverProc algorithm. There is also an entry for the bounded *readwrite* net(B.1), with 11,139 nodes in the K&M tree, taking 530 seconds, a factor 300 slower than CovProc. This is in sharp contrast with my own findings in Section 5.3, where CovProc is a factor 2.5 slower than K&M.

i	V_i	F_i
0	$\varnothing$	$(\langle p1\rangle, \langle p1\rangle)$
1	$(\langle p1\rangle, \langle p1\rangle)$	$(\langle p1\rangle, \langle p2\rangle), (\langle p1\rangle, \langle p6\rangle), (\langle p1\rangle, \langle p7\rangle)$
2	$(\langle p1\rangle, \langle p1\rangle), (\langle p1\rangle, \langle p2\rangle), (\langle p1\rangle, \langle p6\rangle), (\langle p1\rangle, \langle p7\rangle)$	$(\langle p1\rangle, \langle p3\rangle), (\langle p2\rangle, \langle p3\rangle), (\langle p1\rangle, \langle p4, 2p5\rangle),$ $(\langle p6\rangle, \langle p4, 2p5\rangle), (\langle p1\rangle, \langle p2, p5\rangle), (\langle p7\rangle, \langle p2, p5\rangle)$
3	$(\langle p1\rangle, \langle p1\rangle), \cancel{(\langle p1\rangle, \langle p2\rangle)}, (\langle p1\rangle, \langle p6\rangle), (\langle p1\rangle, \langle p7\rangle)$ $(\langle p1\rangle, \langle p3\rangle), (\langle p2\rangle, \langle p3\rangle), (\langle p1\rangle, \langle p4, 2p5\rangle),$ $(\langle p6\rangle, \langle p4, 2p5\rangle), (\langle p1\rangle, \langle p2, p5\rangle), (\langle p7\rangle, \langle p2, p5\rangle)$	$\cancel{(\langle p1\rangle, \langle p4\rangle)}, (\langle p3\rangle, \langle p4\rangle), (\langle p2\rangle, \langle p4\rangle),$ $(\langle p1\rangle, \langle p3, 3p5\rangle), (\langle p4, 2p5\rangle, \langle p3, 3p5\rangle), (\langle p6\rangle, \langle p3, 3p5\rangle),$ $\cancel{(\langle p1\rangle, \langle p3, p5\rangle)}, (\langle p2, p5\rangle, \langle p3, p5\rangle), (\langle p7\rangle, \langle p3, p5\rangle)$
4	$(\langle p1\rangle, \langle p1\rangle), (\langle p1\rangle, \langle p6\rangle), (\langle p1\rangle, \langle p7\rangle)$ $\cancel{(\langle p1\rangle, \langle p3\rangle)}, \cancel{(\langle p2\rangle, \langle p3\rangle)}, (\langle p1\rangle, \langle p4, 2p5\rangle),$ $(\langle p6\rangle, \langle p4, 2p5\rangle), (\langle p1\rangle, \langle p2, p5\rangle), (\langle p7\rangle, \langle p2, p5\rangle),$ $(\langle p3\rangle, \langle p4\rangle), (\langle p2\rangle, \langle p4\rangle), (\langle p1\rangle, \langle p3, 3p5\rangle),$ $(\langle p4, 2p5\rangle, \langle p3, 3p5\rangle), (\langle p6\rangle, \langle p3, 3p5\rangle),$ $(\langle p2, p5\rangle, \langle p3, p5\rangle), (\langle p7\rangle, \langle p3, p5\rangle)$	$\underline{(\langle p3\rangle, \langle p3, p5\rangle)}, (\langle p4\rangle, \langle p3, p5\rangle), (\langle p2\rangle, \langle p3, p5\rangle),$ $\underline{(\langle p1\rangle, \langle p4, 3p5\rangle)}, (\langle p3, 3p5\rangle, \langle p4, 3p5\rangle),$ $\underline{(\langle p4, 2p5\rangle, \langle p4, 3p5\rangle)}, (\langle p6\rangle, \langle p4, 3p5\rangle),$ $\underline{(\langle p2, p5\rangle, \langle p4, p5\rangle)}, (\langle p7\rangle, \langle p4, p5\rangle)$
	at this point, $Accel(F_4)$ accelerates the underlined pairs to pump p5. CovProc$(N, \langle p3, \omega p5\rangle)$ and CovProc$(N, \langle p4, \omega p5\rangle)$ are called.	O_5 is $\{(\langle p3, \omega p5\rangle, \langle p3, \omega p5\rangle), (\langle p3, \omega p5\rangle, \langle p4, \omega p5\rangle),$ $(\langle p4, \omega p5\rangle, \langle p3, \omega p5\rangle), (\langle p4, \omega p5\rangle, \langle p4, \omega p5\rangle)\}$
5	$(\langle p1\rangle, \langle p1\rangle), (\langle p1\rangle, \langle p6\rangle), (\langle p1\rangle, \langle p7\rangle)$ $(\langle p1\rangle, \langle p4, 2p5\rangle), (\langle p6\rangle, \langle p4, 2p5\rangle), (\langle p1\rangle, \langle p2, p5\rangle),$ $(\langle p7\rangle, \langle p2, p5\rangle), \cancel{(\langle p3\rangle, \langle p4\rangle)}, (\langle p2\rangle, \langle p4\rangle),$ $(\langle p1\rangle, \langle p3, 3p5\rangle), \cancel{(\langle p4, 2p5\rangle, \langle p3, 3p5\rangle)},$ $(\langle p6\rangle, \langle p3, 3p5\rangle), (\langle p2, p5\rangle, \langle p3, p5\rangle), (\langle p7\rangle, \langle p3, p5\rangle)$ $\cancel{(\langle p3\rangle, \langle p3, p5\rangle)}, \cancel{(\langle p4\rangle, \langle p3, p5\rangle)}, (\langle p2\rangle, \langle p3, p5\rangle),$ $(\langle p1\rangle, \langle p4, 3p5\rangle), \cancel{(\langle p3, 3p5\rangle, \langle p4, 3p5\rangle)},$ $\cancel{(\langle p4, 2p5\rangle, \langle p4, 3p5\rangle)}, (\langle p6\rangle, \langle p4, 3p5\rangle),$ $(\langle p2, p5\rangle, \langle p4, p5\rangle), (\langle p7\rangle, \langle p4, p5\rangle)$	$\cancel{(\langle p3, p5\rangle, \langle p3, \omega p5\rangle)}, \cancel{(\langle p4, 3p5\rangle, \langle p4, \omega p5\rangle)},$ $\cancel{(\langle p3, p5\rangle, \langle p4, p5\rangle)}, \cancel{(\langle p3, p5\rangle, \langle p4, p5\rangle)},$ $\cancel{(\langle p4\rangle, \langle p4, p5\rangle)}, (\langle p2\rangle, \langle p4, p5\rangle), (\langle p1\rangle, \langle p3, 4p5\rangle),$ $\cancel{(\langle p4, 3p5\rangle, \langle p3, 4p5\rangle)}, \cancel{(\langle p3, 3p5\rangle, \langle p3, 4p5\rangle)},$ $\cancel{(\langle p4, 2p5\rangle, \langle p3, 4p5\rangle)}, (\langle p6\rangle, \langle p3, 4p5\rangle),$ $(\langle p2, p5\rangle, \langle p3, 2p5\rangle), \cancel{(\langle p4, p5\rangle, \langle p3, 2p5\rangle)},$ $(\langle p7\rangle, \langle p3, 2p5\rangle)$

Figure 3.7: Table detailing execution of the CovProc algorithm on the counterexample to MCT.

3.4 Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning

by Pierre-Alain Reynier and Frédéric Servais [RS13]

This paper introduces the *Monotone-Pruning algorithm (MP)*, an improved K&M algorithm with pruning. The pruning is not as agressive as in the MCT algorithm, but this allows the algorithm to be proven correct.

"The main difficulty is to prove the completeness of the algorithm, i.e. to show that the set returned by the algorithm covers every reachable marking. To overcome this difficulty, we reduce the problem to the correctness of the algorithm for a particular class of finite state systems, which we call *widened Petri nets (WPN)*. These are Petri nets whose semantics [is] widened w.r.t. a given marking m: as soon as the number of tokens in a place p is greater than $m(p)$, this value is replaced by ω." The algorithm is proven to be terminating and correct on WPN and standard Petri nets.

Their definition of a coverability set matches ours almost literally, as it is a set C of ω-markings such that:

"1) for every reachable marking m of $\mathcal{N}$, there exists $m' \in C$ such that $m \leqslant m'$,

2) for every $m' \in C$, either m' is reachable in $\mathcal{N}$ or there exists an infinite strictly increasing sequence of

reachable markings $(m_n)_{n\in\mathbb{N}}$ converging to m'."

"The K&M Algorithm uses comparisons along the same branch to stop exploration (test of Line 5), that we call *vertical pruning*. We present in this section our algorithm which we call Monotone-Pruning Algorithm as it includes a kind of *horizontal pruning* in addition to the vertical one. We denote this algorithm by MP. It involves the acceleration function Acc used in the Karp and Miller algorithm. However, it is applied in a slightly different manner.

Algorithm 2 Monotone Pruning Algorithm for Petri Nets.

Require: A Petri net $\mathcal{N} = (P, T, I, O, m_0)$.
Ensure: A labelled tree $C = (X, x_0, B, \Lambda)$ and a set Act $\subseteq X$ such that $\Lambda(\text{Act}) = \text{MCS}(\mathcal{N})$.

1: Let x_0 be a new node such that $\Lambda(x_0) = m_0$;
2: $X := \{x_0\}$; Act $:= X$; Wait $:= \{(x_0, t) \mid \Lambda(x_0)\,[\,t\,\rangle\,\cdot\}$; $B := \varnothing$;
3: **while** Wait $\neq \varnothing$ **do**
4: Pop (n', t) from Wait.
5: **if** $n' \in$ Act **then**
6: $m :=$ Post$(\Lambda(n'), t)$;
7: Let n be a new node such that $\Lambda(n) =$ Acc$(\Lambda(\text{Ancestor}_C(n') \cap \text{Act}), m)$;
8: $X := X \cup \{n\}$; $B := B \cup \{(n', t, n)\}$;
9: **if** $\Lambda(n) \not\leqslant \Lambda(\text{Act})$ **then**
10: Act $:=$ Act $\setminus \{x \mid \exists y \in \text{Ancestor}_C(x)$ s.t. $\Lambda(y) \leqslant \Lambda(n) \wedge (y \in \text{Act} \vee y \notin \text{Ancestor}_C(n))\}$;
11: Act $:=$ Act $\cup\{n\}$; Wait $:=$ Wait $\cup\{(n, u) \mid \Lambda(n)\,[\,u\,\rangle\,\cdot\}$;
12: **end if**
13: **end if**
14: **end while**
15: Return $C = (X, x_0, B, \Lambda)$ and Act."

"...nodes of the tree are partitioned into two subsets: active nodes, and inactive ones. Intuitively, active nodes will form the minimal coverability set of the Petri net, while inactive ones are kept to ensure completeness of the algorithm."

In this algorithm, Ancestor_C denotes the ancestors of a node, including itself. Act is always kept minimal this way: if a node is covered by Act it will not pass line 9, and if a node greater than some active node $\overline{n}$ is found, it will be removed in line 10 by $x = y = \overline{n}$. Only active nodes need be used for acceleration, as any inactive ancestors have been deactivated because a greater ancestor exists. Inactive nodes merely exist as candidates for y in line 10, and even then only a subset of them needs to be kept:

(a) $y \notin \text{Ancestor}_C(n)$ (b) $y \in \text{Ancestor}_C(n) \cap \text{Act}$

Figure 2. Deactivations of MP Algorithm.

Figure 3.8: Original figure detailing MP algorithm.

On the left of Figure 3.8, if a node n is found such that another node is *horizontally pruned*, then that node and all its descendants will not pass line 5. None of these nodes are in the MCS, nor will they be used for accelerations, they can be discarded. On the right of Figure 3.8, if a node n is found greater than an ancestor y of n, then all its descendants are removed in line 10. But in line 11, n is added to Act. Only the deactivated nodes on the path from y to n are relevant, as they can still be used to deactivate n and its descendants.

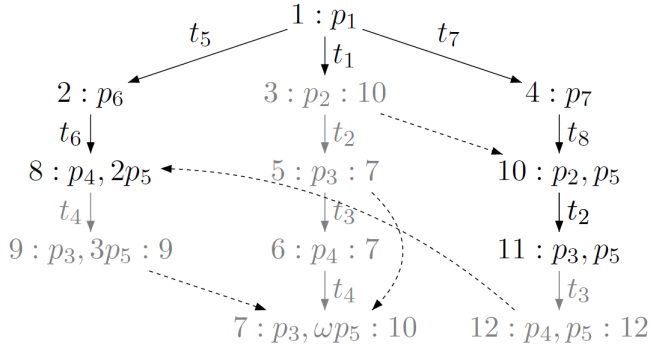


Figure 8. An execution of the MCT Algorithm on $\mathcal{N}_{cex}$.

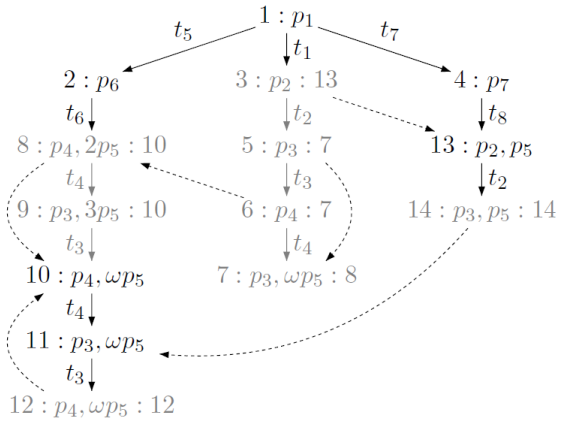


Figure 9. An execution of the MP Algorithm on $\mathcal{N}_{cex}$.

Figure 3.9: Original figures detailing the difference between MCT and MP on the counterexample to MCT.

We can see that because of these mechanisms, the MP algorithm is able to handle the counterexample net $\mathcal{N}_{cex}$, as shown in Figure 3.9. In MCT, because node 6 is deactivated by node 7, it cannot be used by node 8 to deactivate node 7. Node 9 is then deactivated by node 7 as soon as it is discovered. This pitfall is avoided by the MP, as deactivated nodes can be used to deactivate their descendants. When node $n = 8$ is discovered, Line 10 of the algorithm is used with $x = 7$, $y = 6$, and 7 is deactivated. Node 9 is now uncovered by Act, and is not deactivated on discovery.

The order of the elements in Wait is not fixed, any element can be popped in Line 4. They prove that the order does not matter for the correctness of the algorithm, so depth-first, breadth-first, or any order can be used.

The algorithm is then proven correct, and compared to the K&M algorithm, the MCT aclgorithm, and the CoverProc algorithm of the previous paper [GRB10]. The nets used to compare them are a subset of the nets used in the previous paper as well. Two versions of the MP algorithm are used, a breadth-first and a depth-first version.

The algorithms were implemented in Python, and compared in runtime and number of elements created. Furthermore, the CoverProc algorithm wasn't implemented, the data for its runtimes seem to be copied from its paper. As there are several years inbetween these papers, they might have used inferior hardware for computing these times, so it is ill-advised to draw conclusions from these numbers. However, as in my

own results, CoverProc is several orders of magnitude slower than the MP algorithm, which itself has speed similar to MCT, orders of magnitude faster than K&M. The BFS approach is overall slightly faster than the DFS, but this is only significant in the *bounded* net "mesh2x2".

In summary, the MP algorithm is fast, correct, and the order of exploration is completely free, allowing for different exploration strategies. Indeed, in my own implementation, it uses the "most tokens first" approach from the next paper.

3.5 Old and new algorithms for minimal coverability sets

by Antti Valmari and Henri Hansen [VH14]

The final paper we review introduces some new ideas. The reason previous algorithms were incorrect, or why they required a lot of checking, was because they prune futures. If a marking covers an old one, not only that marking but (a subset of) its descendants are deactivated as well. In this publication, a simpler algorithm that lacks future pruning is presented and proven correct. It is demonstrated, using examples, that *neither approach is systematically better than the other*.

This paper has a definition of MCS equivalent to ours: "*Let* $M_1, M_2, \ldots$ *be* ω*-markings such that* $M_1 \leqslant M_2 \leqslant \ldots$. *Their* limit *is the* ω*-marking* $M = \lim_{i\to\infty} M_i$ *such that for each* $p \in P$*, either* $M(p) = M_i(p) = M_{i+1}(p) = M_{i+2}(p) = \ldots$ *for some* i*, or* $M(p) = \omega$ *and* $M_i(p)$ *grows without limit as* i *grows.*" Note that ω tokens are allowed in the markings M_i, as long as they are repeated in every M_i thereafter.

"*Let* $\mathcal{M}$ *be a set of* ω*-markings. Then* M *is a* limit *of* $\mathcal{M}$ *if and only if there are* $M_1 \in \mathcal{M}, M_2 \in \mathcal{M}, \ldots$ *such that* $M_1 \leqslant M_2 \leqslant \ldots$ *and* $M = \lim_{i\to\infty} M_i$. *. . . Let* $\mathcal{M}$ *be a set of markings and* $\mathcal{M}'$ *a set of* ω*-markings. We define that* $\mathcal{M}'$ *is a* coverability set *for* $\mathcal{M}$*, if and only if*

1. *For every* $M \in \mathcal{M}$*, there is an* $M' \in \mathcal{M}'$ *such that* $M \leqslant M'$.
2. *Each* $M' \in \mathcal{M}'$ *is a limit of* $\mathcal{M}$.

A coverability set is an antichain*, if and only if it does not contain two* ω*-markings* M_1 *and* M_2 *such that* $M_1 < M_2$. *[. . .] Each set* $\mathcal{M}$ *of markings has a coverability set that is an antichain. It is finite and unique. It consists of the maximal elements of the limits of* $\mathcal{M}$. *[. . .] Each set of markings has precisely one minimal coverability set.It is finite. It is the antichain coverability set.*"

Their algorithm is unnamed, and I will refer to it in later sections as *the HanVal algorithm*. The algorithm tracks ALL *found markings* in a set F, the *active markings* in a set A, a *workset* of unprocessed pairs W, and the tree itself, by assigning a *back pointer* $M.B$ to every marking M. The set F is never trimmed, and exists

so no subtree is explored twice. F is implemented as a hashing table or similar data structure, so that it is fast to test whether a marking has already been found. When pumping for ω tokens, in the "original KM algorithm" all ancestors of M are compared to M, and add tokens to a marking M', which then replaces the label M in the tree. But ancestors might also be smaller than this marking M'! Thus, this algorithm uses *repeated history scanning* in the Add-ω routine:

As shown in Figure 3.11, instead of pumping only p_1 because $(2,1,2) > (1,1,2)$, the result marking $(\omega,1,2)$ checks the history again. Because $(\omega,1,2) > (3,1,1)$, p_3 is pumped as well, so that the final result marking is $(\omega,1,\omega)$. A notable difference with previous algorithms is that the ancestors used to pump need not be active. While checking ancestors takes longer, the total amount of nodes can go down, as the pumping sequence need not (partly) be fired again from the pumped node.

Another new technique used is *history merging*. Instead of a single back pointer $M.B$, there is a list of pointers. If a marking M' is encountered that is already in F, M is added to its back list. This requires some adjustments in the Add-ω function, as all ancestors in what is now a subgraph above M' must be compared to M'. An example of how this is helpful is shown in Figure 3.12, as the new marking $(0,1,1,0)$ is now larger than two of its predecessors, and can be pumped to $(0,\omega,\omega,0)$.

The active set A is kept minimal, so that A is the MCS upon termination. Cover-check(M') checks if M' is covered by any element in A, and if not, which elements of A are covered by M'. These elements are then removed, and M' is added to A. Any pairs in W whose first element was just removed from A is also removed from W. Finally, the new marking is added to F, A, and all pairs (M',t) are added to W.

This algorithm is proven to terminate and be correct for any order of elements of W. Instead of adding $\{M'\} \times T$ to W, the practical way is to let W simply be a set of markings, and to have each marking M have the property $M.next\_tr$ denoting the next transition to check. When a marking is removed from A, we can now remove its "associated pairs" from W easily, by setting $M.next\_tr$ to an end value.

The paper compares Breath-First Search(which has the added benefit of making W a queue), Depth-First

```
1  F := {M̂}; A := {M̂}; W := {M̂} × T; M̂.B := nil
2  while W ≠ ∅ do
3      (M, t) := any element of W; W := W \ {(M, t)}
4      if ¬M [ t ⟩ · then continue
5      M' := the ω-marking such that M [ t ⟩ M'
6      if M' ∈ F then continue
7      Add-ω(M, M')
8      if ω was added then if M' ∈ F then continue
9      Cover-check(M')        // may update A and W
10     if M' is covered then continue
11     F := F ∪ {M'}; A := A ∪ {M'}; W := W ∪ ({M'} × T);M'.B:= M
```

Figure 3.10: The basic coverability set algorithm.

Add-$\omega(M, M')$

```
1 last := M; now := M; added := False
2 repeat
3     if now < M' ∧ ∃p ∈ P : now(p) < M'(p) < ω then
4         added := True; last := now
5         for each p ∈ P such that now(p) < M'(p) < ω do
6             M'(p) := ω
7     if now.B = nil then now := M else now := now.B
8 until now = last
```

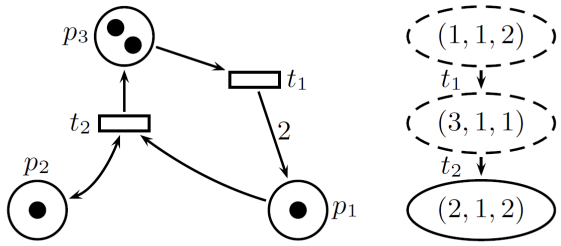


Fig. 3. The advantage of repeated scanning of history

Figure 3.11: Original figures detailing repeated scanning of history.

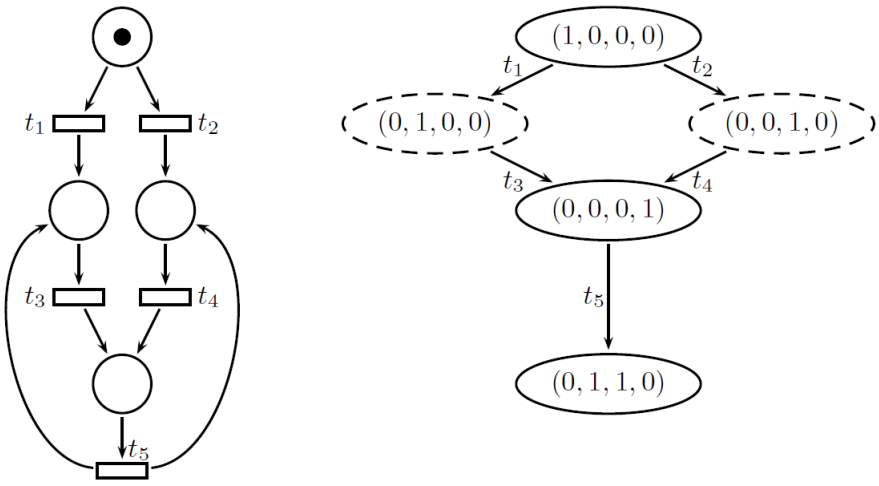


Fig. 4. How history merging helps

Figure 3.12: Original figure detailing the advantage history merging.

Search(which makes W a stack), and the new *Most Tokens First search*. Most-tokens first sorts the markings M by amount of ω tokens: $|\Omega(M)|$, and then by amount of supply in the non-ω places: $\sum_{p \in \overline{\Omega}(M)} M(p)$. This greedy heuristic is based on the ideas that after pumping, the subtree at M' will likely passivate a large number of active markings, and should thus be explored first. Also, any pumping sequence will likely have an increasing number of tokens in it, so those with most tokens are most likely to lead to pumping sequences.

"In our measurements (see Section 6), breadth-first was never clearly the fastest but was often clearly the slowest. So we do not recommend breadth-first." Instead, it is shown that using DFS or MTF with history merging gives almost the same advantages as future pruning does:

"**Theorem 3**. *Let the construction order be depth-first and history merging be applied. Assume that* $M_0 [t_1 \cdots t_n \rangle^\omega M_n$ *and* $M_0 < M'_0$. *Assume that all transitions along the path* $M_0 [t_1 \cdots t_n \rangle^\omega M_n$ *were found before* M'_0. *After finding* M'_0, *the algorithm will not fire transitions from* M_n, *unless* $M'_0 [t_1 \cdots t_n \rangle M_n$." [VH14]

Here, $[t \rangle^\omega$ denotes that accelerations may have happened after firing the transitions. An identical theorem is given for most-tokens-first order. Intuitively, There is no need to passivate any descendants M_n of M_0, because they will be passivated once $M'_n | M'_0 [t_1 \cdots t_n \rangle^\omega M'_n$ is found. If $M_n = M'_n$, this passivation would need to be undone as well. They conclude that future pruning is merely a nuisance for this algorithm. This includes passivation/removal of nodes in a pumping cycle.

They specify this is one of the major reasons they suspect the HanVal algorithm to outperform the previous MP-algorithm, as that one "requires checking whether the new ω-marking M strictly covers any element in F (excluding the history of M). This is a disadvantage, because otherwise checking coverage against A would suffice, A may be much smaller than F, and checking coverage is expensive." In the basic MP-algorithm, Figure 3.8, deactivated nodes are indeed kept in the tree C, and a naive approach of line 10 could prove costly. However, further reading shows that in their implementation, only the deactivated nodes in a pumping cycle are kept, while other deactivated nodes are removed entirely: A is hardly smaller than F.

An objective analysis of the HanVal algorithm follows, where they show two small Petri nets, and how either DFS or MTF can outperform the other. On some slightly more standardized benchmarks "[MTF] was often both the fastest and constructed the smallest number of ω-markings."

Finally, they show some of their test results, as shown in Table 3.1. BFS, DFS and MTF were ran on six benchmarks (fms, kanban, mesh2x, mesh3x2, multipoll[sic] and pncsacover) and shown is their total number of constructed distinct ω-markings, that is, $|F|$. As transitions on markings are fired in order, they "ran each experiment with transitions tried in the order that they were given in the input and in the opposite order", sometimes with a dramatic impact on the result. *"This acts as a warning that numbers like the ones in the table are much less reliable than we would like."* Next to these numbers are results from the MP-algorithm's paper, showing "number of elements passed in the waiting list". The MP-algorithm however, is not only interested

model	$\|A\|$	most tokens f.		depth-first		breadth-first		[6]
fms	24	63	53	110	56	421	139	809
kanban	1	12	12	12	12	12	12	114
mesh2x2	256	479	465	774	455	10733	2977	6241
mesh3x2	6400	11495	11485	8573	10394			
multipoll	220	245	234	244	244	507	507	2004
pncsacover	80	215	246	284	325	7122	5804	1604

Table 3.1: Results from the paper.

in the MCS but also the tree created. As such, most elements of the MCS are the label of many different nodes. It is unclear why these results are posted side-by-side, for they are useful for comparing neither memory use nor runtime.

I was impressed by the heuristics used in MTF, and as the order of the workset is adjustable in the other algorithms too, have incorporated it in the runtime-prioritizing variant of the algorithms.

Chapter 4

Theoretical analysis

While these algorithms all calculate the Minimal Coverability Set, there are some important differences between the algorithms. The properties I will compare are:

- Correctness. Does the program calculate the MCS as claimed?
- Speed.
- Memory consumption.
- Amount of parallelism possible. A subjective measure, as I merely show the steps I did in my implementation.
- Firing Sequence detection. A major reason for calculating a SCG of a graph is to determine which firing sequences are possible from the root. While unbounded nets yield an infinite number of possible firing sequences, they can still be described by some form of expression. For example in Figure 2.4: "$(1, 0, \omega)$ is reachable by $\sigma_n = (t_1 t_2)^n$, as $\hat{M}\,[\,\sigma_n\,\rangle\, M_n \geqslant (1, 0, n)$". After calculating the elements of the MCS, is it possible to construct a firing sequence for each element?
- Interconnectivity. The algorithms' ability to show what elements of the MCS can reach each other. Is the entire MCS reachable from any of its elements, can you split the MCS into components? If connected, what is the sequence to get from configuration to configuration?

This last one is not as trivial as it seems, as shown in Figure 4.1. From the initial state, both t_1 and t_4 are fireable, both leading to a marking covered by $(\omega, \omega, 0)$. Firing t_4 is a 'dead end' though, the interconnectivity is not always clear from a shallow inspection of the MCS.

4.1 Karp and Miller

The original Karp&Miller algorithm [KM69] is very slow and uses a lot of memory, but it preserves a lot of data. No node is ever edited after it is created, so this algorithm is very *parallelizable*, with a large read-only tree shared between threads.

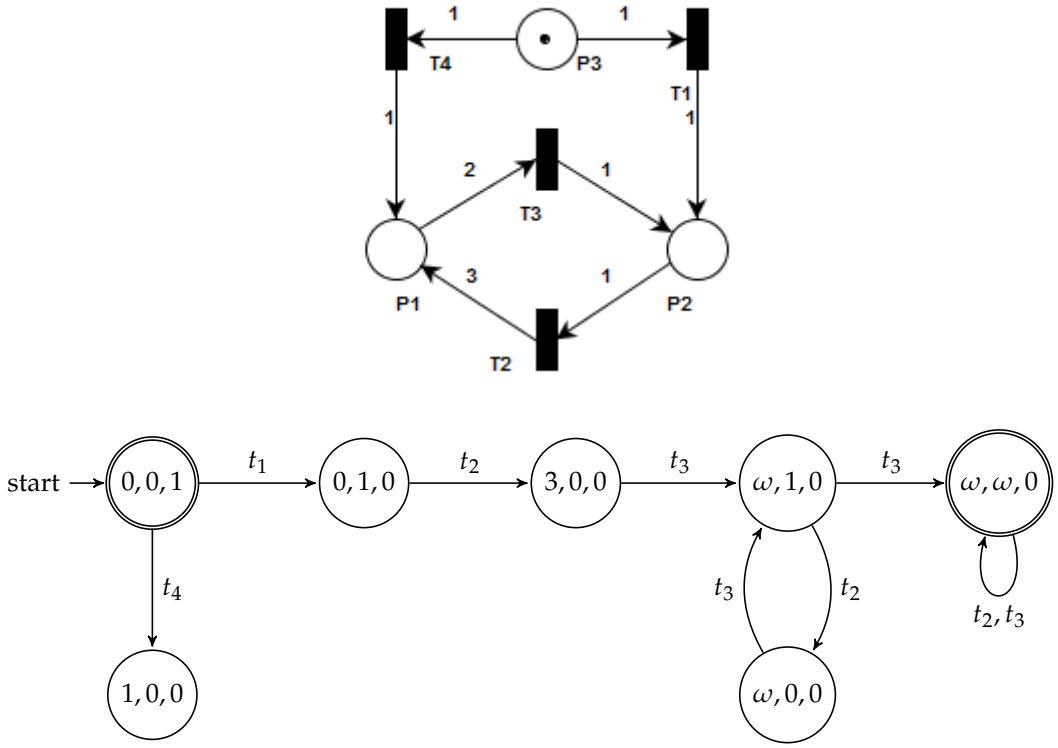


Figure 4.1: A graph and its coverability graph.

By examining the arcs `arc(n,t,n')` the algorithm stores, it is possible to construct all the fireable sequences from any node in the graph, as there are many repeated subtrees. The sequence of any path from M to one of its descendants $M' \geqslant M$ can be fired arbitrarily many times from M. The interconnectivity is likewise obvious, as there is a firing sequence leading from M to M' iff there is a path in the tree from a node labeled M to one labeled M'.

4.2 MCT/MCG

To combat the runtime and memory costs, the paper by Finkel [Fin91] suggests a modification of the K&M algorithm *"...to obtain an algorithm which directly computes the minimal coverability graph without computing the whole Karp-Miller graph."* While the algorithm is fast and does not take much memory, it is not always correct, and can rarely produce an error depending on the order the unfinished nodes are explored. [GRB10]

The MCT algorithm will not explore nodes if a marking covering that node has already been explored. This covering node does not have to be an ancestor, but can be anywhere in the tree. Retroactively, any new node added to the tree removes any nodes it covers, and their subtrees. If a node covers one of its ancestors, it replaces it. With these modifications, we lose the relationship between arcs and firing a transition, with no clear way to see either a sequence to get to a node nor which nodes can reach each other. The parallelism takes a minor hit, as two comparable markings could be added to the workset at the same time, but this is

rectified by another check whether a covering element already exists.

4.3 CoverProc

The CoverProc algorithm [GRB10] uses pairs of markings (m_1, m_2) to indicate there is some sequence, possibly including accelerations, to get from m_1 to m_2. The algorithm was devised with correctness in mind, and is the largest and slowest of the non-K&M algorithms. No transitions are stored anywhere, so no firing sequence detection.

The pairs are explored in a recursive algorithm, essentially a BFS. Sets of visited and frontier pairs are generated inductively, and on each iteration their *minimized, Flattened* sets are compared, i.e. the set of only the second markings in the pairs, with smaller markings removed. If this set doesn't change in an iteration, the algorithm is finished and returns the sets of visited pairs. Each separate set generation can be parallelized, but the additional waiting and overhead has not shown any actual speedups in my tests.

Whenever an acceleration is found, the accelerated node is used as root in a recursive call to CoverProc, so it is DFS wrt accelerations. The result of this recursive call is used in the layer before it as the *oracle*, removing smaller pairs wrt the special $\sqsubseteq$ relationship. For any two pairs, $p_1 \sqsubseteq p_2$ if the pairs in p_2 are greater than those in p_1, and p_2 has a greater *growth* (i.e., $m_2 - m_1$) than p_1.

Ideally, this should remove any pairs not needed for calculating the MCS, and prevent accelerations and costly recursive calls. But the recursive call also stops when the Flattened set does not change, and returns an "incomplete" set of pairs. Consider the net in Figure 4.2. We can immediately see that the MCS contains the markings $p_1\omega p_3$ and $p_2\omega p_3$ and that these markings are reachable from each other. After firing $t_1t_3t_4$, the visited pairs contain (p_1, p_1p_3), and a recursive call is started with $p_1\omega p_3$ as root. After firing t_3t_4, the recursive call ends, but it does not return $(p_2\omega p_3, p_2\omega p_3)$. As a result, the pair (p_2, p_2p_3) generated from $t_2t_4t_3$ is not covered (wrt $\sqsubseteq$) and does a recursive call as well.

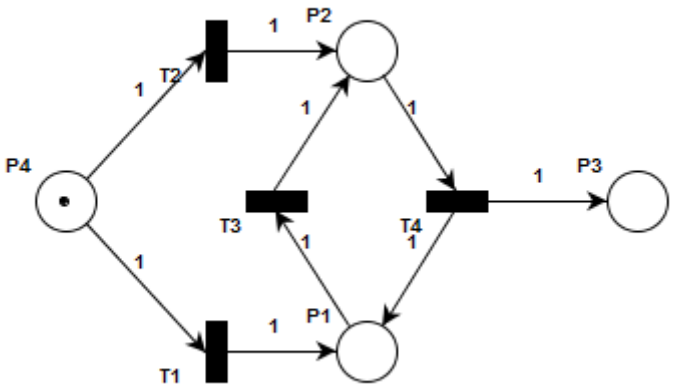


Figure 4.2: A bad net for CoverProc.

This also highlights the lack of interconnectivity. When an acceleration $M < M'$ is found, only the pair (M, M^ω) is added to the visited pairs, but not any other pair (N, M^ω), even if M was reachable from N. In the example, $(p_1, p_1\omega p_3)$ exists, but neither $(p_4, p_1\omega p_3)$ nor $(p_1, p_2\omega p_3)$ do.

4.4 Monotone Pruning algorithm

Reynier and Servais had the approach of doing both horizontal and vertical pruning with the Monotone Pruning Algorithm. It uses a visited set and a working set, which can contain active and inactive nodes. Only active nodes can deactivate other nodes, preventing the deactivation loop that was possible in the MCT algorithm. The algorithm is proven correct. The flexible part of the algorithm is the way to pick pairs (M, t) from the working set. The paper compares BFS, DFS, and 'most tokens first' approaches, considering the last one superior in most cases. Whichever way chosen, the speed is comparable or superior to the speed of MCT, with minimal memory cost. It is also easy to run in parallel, as the only side effect of the race condition is too many nodes in the active set, which does not interfere with the correctness. Transitions are stored in the tree, and nodes used in an acceleration are deactivated, not removed, so the accelerating sequence is visible as well.

Interconnectivity is not readily visible from the tree. Let $M\,[\,t\rangle\,N\,[\,\ldots\rangle\,N'$, and $M'\,[\,t'\rangle\,N$. Naturally, N' is also reachable from M', but when the second node labelled N is found, a marking equal to it (and thus covering it) has already been found, and it is deactivated. If we transform the tree into a reachability graph by identifying nodes with the same label, the path from M' to N' emerges, and interconnectivity is as simple as checking for existence of a path.

4.5 HanVal

Hansen and Valmari went a different way, pruning not more but less than MCT. The unnamed algorithm (called HanVal by me) utilizes a working set, with pairs (M, t) of markings and transitions (possibly optimized away in the implementation). It uses a pruning system prunes similar to MCT, but only deactivates nodes when a covering node is being explored, not when it is discovered. The correctness is proven in the paper, and its speed and memory are comparable to the MP algorithm. The algorithm is able to run in parallel quite well, as the only critical part of the algorithm is Cover-check. Even that part can be run in parallel a bit, as it is read-only if a covering marking is found.

The firing sequence detection is harder, because of history merging. Even when a back pointer would store the transition used, the execution of Add-ω depends on the entire history of a node *n at that moment*. In the final graph, this node can have 'more' history, if some node merged its history with an ancestor of n. The good side of this is that interconnectivity is as easy as checking whether a path exists in the tree.

4.6 Summary

Name	Correct	Speed	Memory	Parallelizable	Sequence detection	Interconnectivity
K&M	Y	Slowest	Most	All	Y	Y
MCT	Mostly	Fast	Little	Mostly	N	N
CoverProc	Y	Slow	Much	Little	Y	N
MP	Y	Fast	Least	Mostly	Y	minor work
HanVal	Y	Fast	Least	Mostly	N	Y

While these criteria are sometimes important, the biggest factors are still execution time and memory cost. In the next chapter we will compare implementations of the algorithms on some Petri nets for better comparison.

Chapter 5

Empirical evaluation

5.1 Program

Because theoretical investigation of the algorithms can only go so far, the only way to really compare the algorithms is to implement them and use them on some Petri nets. To this end I implemented various versions of each of the algorithms we have discussed so far. The algorithms were then tested on a series of benchmarks, not just the most favorable Petri nets for each algorithm, as is usually done in the algorithms' own presentations. Of these benchmarks, four are bounded nets, seven are unbounded.

The benchmarks used are described in Table 5.1, and the subsequent figures detail each algorithms' execution time. Tests were ran on more bounded nets, but execution times were less than a millisecond, even for KarpMiller, so these nets were not useful for comparison. As a result of this benchmark set, we can also speak of *the number of accelerations* in a net, as there are no mutually exclusive ω tokens such as in Figure 2.6. The .spec files do not offer an insightful first-glance view of the nets, so there are images of the Petri nets in Appendix B, made with the program PIPE[1].

Unfortunately, most of these unbounded benchmarks do not actually have an unbounded Petri net graph, but merely have an initial marking containing at least one ω token. This token is then multiplied through the transitions, sometimes even resulting in an MCS of the sole marking $\omega^{|P|}$. Would these nets have a standard initial marking, no ω token would ever come to be. Only one benchmark, "pncsacover", has a normal accelerating sequence: $M\,[\,\sigma\,\rangle\,M'$ where $M' > M$ and $\Omega M = \varnothing$.

Runtime and memory used are subjective factors, as they are influenced by both my specific implementation, and the fact that the implementation language used, C#, is garbage collected instead of having exact memory management. However, the amount of memory units taken, be it nodes, pairs or markings, is still an objective measure of the algorithms' effectiveness.

The program was executed on an Intel i5-2500K CPU at 3.3 GHz, 16GB RAM, running Windows 7 Home

[1] http://pipe2.sourceforge.net/

Premium. The 32-bit version of the program was used, so that algorithms were stopped when their used memory exceeded 4GB.

There are three versions of each algorithm: the *Simple* versions are just that, the algorithms implemented in the most straightforward way. This includes implementing a $\forall x \in \{\ldots\}$ as a `for` statement iterating that set, when this may not be a good way to do it. The Simple versions keep all data, the complete trees, transition names, etc., if the algorithm did that. There is the *Speed* versions, which focus on calculating the MCS. No redundant data is stored, all data is thrown away as soon as it isn't needed for the MCS. Most algorithms use the most-tokens-first approach that Hansen and Valmari introduced, as the algorithms allow for any order, and this MTF proved superior in early tests. Lastly there are *Parallel* versions. Most algorithms are described as single-threaded programs, but even home computers have quad core CPUs. Most algorithms were suited for a Worker approach, where several Worker threads(scaling to the amount of available cores) execute the algorithm in parallel, with some care taken to avoid race conditions and incorrect executions. Only CoverProc was not suited for this, which instead tries to calculate the new generation of sets in parallel as much as possible. The HanVal algorithm does not have Simple and Speed, but DFS and MTF versions, to compare my results to those in its paper.

The program is run as `PetriTester <TestSize> [<timeoutms> [<Folder>]]`. It will check the folder for files ending in ".spec", and tries to interpret them as Petri nets, or 'benchmarks'. It runs the algorithms in a random order on the benchmarks in a random order, until each combination has run TestSize amount of times. It stores the results in the file "results.dat". For each execution it stores the results:

- the amount of milliseconds taken
- the amount of ticks taken, a slightly more precise unit
- the amount of memory units in use at program termination
- the maximum amount of memory units in use at any time during execution
- the amount of elements in the calculated MCS
- the first 20 elements of the MCS

Subsequent program executions will sort the "results.dat" file to a more readable format if any algorithm execution failed, and add result to the file if a greater TestSize was given. No previous results are erased.

- The main program is found in `Program.cs`. It builds a list of `PetriNet` objects that it reads from the `.spec` files, and creates a list of `Grapher` objects that are the implementations. It then reads the "results.dat" file for previous results, and runs the Grapher-PetriNet that aren't found (enough). The combination is run in a separate thread, so that it can timeout after 10 minutes (or the given timeout parameter). It saves the results back into the "results.dat" file after each combination.

- The file `Common.cs` holds the common elements all algorithms share. The `Grapher` base class is described here, requiring at least a `Run` method and a `FinalMarkings` property to be implemented by subclasses. The algorithms are subclasses of Grapher.
  The classes `PetriNet`, `Transition` and `Marking` are here as well.

- `DataStructures.cs` holds the `MTFHashSet`, a set with fast operations for adding a marking and checking existence, while still having a fast operation for retrieving the most-tokens marking. Also holds `ConcurrentMTFHashSet`, which is thread-safe. Both of these use $|P| + 1$ lists internally, each holding markings with a different number of ω symbols, and each ordered by the supply in the non-ω places.
- `FileHelper.cs` and `RunHelper.cs` hold methods for reading files, saving results, and concurrent execution. The subfolders hold versions of the algorithms of that name.

The Benchmarks used are originally from the paper by Geeraerts, and can be found at their university's website.[2] Most of them have also been used in the paper by Reynier and Servais, and in the paper by Hansen and Valmari. The Petri nets used can have self-loops, as the transitions are described only as "(places required for action) $\rightarrow$ (effect of action)". There is currently no support for actions that require a specific range of supply in input places, nor for transfer or reset arcs. Adding additional benchmarks is as simple as adding them to the folder, provided they follow the same syntax. Adding algorithms requires changing the code, as they are implemented as Grapher objects. While it would be possible to implement this as loading `.dlls`, this seemed too much work for a probably unused function.

5.2 Implementations

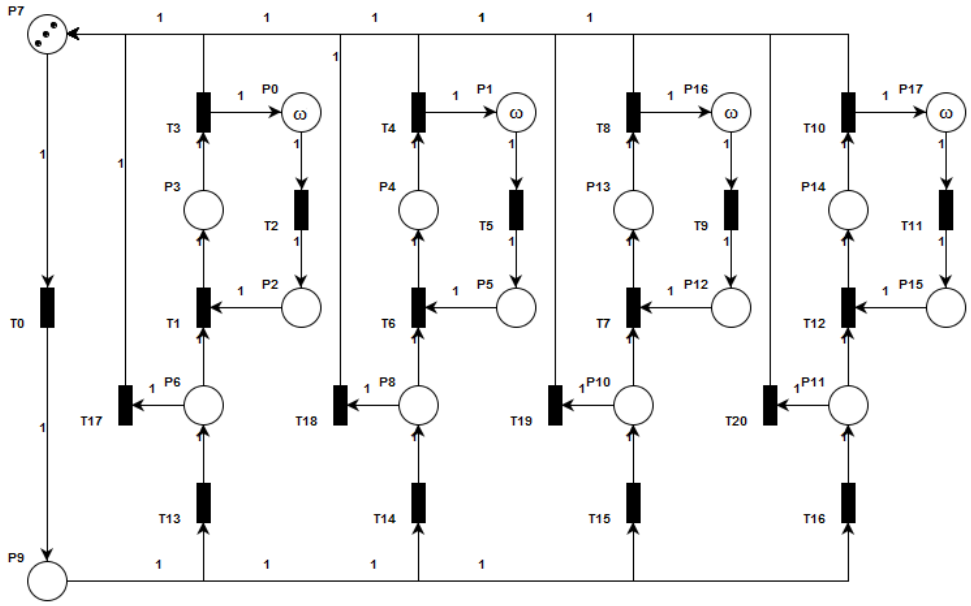
Most Simple versions use C#'s `HashSet` data structure, which allows for fast enumeration of the set, addition and removal elements, and checking whether an element already exists. The elements are ordered by their hash, resulting in a pseudo-random ordering of Markings, so that iterating the elements is not specifically BFS nor DFS. Hashes are calculated only once each marking and are then memorized.

5.2.1 KarpMiller

The Karp&Miller [KM69] algorithm (A.2) retains all data about firing sequences, only checks for duplicate markings in the ancestors of each node, and never trims or prunes in any way. Inefficient for bounded nets, as this tree can have a height of $|\mathcal{R}|$ and a breadth 'proportional to its concurrency'. It's worse for unbounded nets.

The Simple version iterates through the unfinished nodes by calling `HashSet.First()`, a pseudo-random order based on the hashes for specific nodes. Nodes use the hash of their label, so adding an identically labelled markings costs time linear to the amount of identically labelled markings already in the unfinished nodes set. The Speed version goes for a DFS, as the order of exploration does not influence the resulting tree. This has the advantage of lacking parent pointers: a node becomes a parent when explored, and stops being one when this node is backtracked from. This is implemented with a Stack of markings, when a node is done the next element can be popped from the stack. The Parallel version goes for BFS, as there are fewer collisions between worker threads that way, and it's not possible to omit parent pointers.

[2] http://www.ulb.ac.be//di/ssd/ggeeraer/eec/



The benchmark multipool.

The Speed and Parallel version are only minimally faster than the Simple version, as it still has to compute the entire tree. This is apparent on e.g. Petri net 'multipool' (Figure B.7): all four transitions t_2, t_5, t_9 and t_{11} lead to an acceleration immediately, but there are already $4! = 24$ different ways to do this, and any combination of the other transitions can be fired before/after as well. The execution of Speed was stopped after two hours, with no end in sight. All elements of the MCS had already been found because of the DFS approach, but there was still an enormous tree to check.

5.2.2 MCT

The MCT [Fin91] algorithm (A.3) calculates the MCS with two separate procedures, simply named *minimal_coverability_tree* and *minimal_coverability_graph*. The Simple version keeps this structure, and iterates through the unfinished nodes by pseudo-random Hashcode order. Nodes with already found markings are still added to the tree, as is all transition information. After constructing the tree, it runs MCG to identify equivalent Nodes.

The Speed and Parallel versions use the MTF order instead, and do not store any transition data. Furthermore, no duplicate nodes are stored in the tree, so that it is not needed to identify them in MCG. This also allows for faster coverage checking, as there are less Nodes to check against. However, no attempt was made to combat the error, so the Parallel version is still quite prone to giving the wrong answer. Sometimes this rarely happens, but on some nets this happens consistently.

5.2.3 CovProcGraph

The CovProcGraph [GRB10] implementation (A.4) uses HashSets, where hashes of pairs are dependant on both their markings, as no datastructure is fit for quick $\sqsubseteq$ checking. The only way to determine the maximum elements in a set wrt. $\sqsubseteq$ is pairwise comparison of all of them, but we can use our knowledge of the sets for some optimizations.

When determining $V_i := \text{Max}^{\sqsubseteq}(V_{i1} \cup F_{i1}) \setminus \downarrow^{\sqsubseteq}(O_i)$, some elements need not be checked. Both V_{i-1} had elements covered by O_{i-1} removed from them in the previous iteration, so if O_i is unchanged, this check need not be made again. Likewise, F_{i-1} does not need to compare elements with O_i if it is unchanged, but still has to compare them to V_i.

When determining $C = \max^{\sqsubseteq}(A \cup B)$, where both sets A and B already contain maximal elements, then $\forall a \in A, b \in B : a \sqsubseteq b$ implies:

- $a \notin C$
- $b \in C$, because if there were an element $a' \in A$ s.t. $b \sqsubseteq a'$, then because the transitivity of $\sqsubseteq$ implies that $a \sqsubseteq a'$, conflicting with the maximality of A.
- There is no need to check for $b' \neq b \in B$ that $b' \sqsubseteq a$, because transitivity implies $b' \sqsubseteq b$, conflicting with the maximality of B.

These considerations speed up the process of calculating $\max^{\sqsubseteq}$, but CovProc remains the slowest algorithm by a landslide, apart from KarpMiller.

5.2.4 ReySer

The algorithm (A.3) by Reynier and Servais [RS13] does not impose an order in which the set of waiting nodes should be processed, and compares a Depth First Search(DFS) and Breadth First Search(BFS) approach. The Simple version uses a DFS order while the Speed version uses MTF, and the Parallel version uses MTF as well. In every version, the algorithm is still:

1. Initialize active and waiting set with pairs of initial node M and every transition t.
2. Pop a pair (M, t) from the waiting set, or quit if there is none.
3. If t is not fireable jump to 2.
4. Fire t from M to obtain M', then accelerate it to M'' by using the ancestors of M'. If any active node covers M'', jump to 2.
5. Remove from active and waiting those nodes for which there is an ancestor(including the node itself) that is covered by M'' and is either active or is NOT an ancestor of M'. Add M'' to active and (M'', t) to wait for every transition t. Jump to 2.

The Speed/Parallel version use an enhanced version of this line:
10: Act := *Act* $\setminus \{x \mid \exists y \in Ancestor_C(x)$ *s.t.* $\Lambda(y) \leqslant \Lambda(n) \wedge (y \in Act \vee y \notin Ancestor_C(n))\}$.
With a naive approach, this could cause a lot of useless checking, as any node is checked as candidate y for every one of its descendants x. Instead, all nodes n are checked for being a possible y once, and those that match are removed, with all their descendants.

5.2.5 HanVal

The unnamed algorithm (A.6) by Hansen and Valmari [VH14] returns to a K&M-like algorithm without any pruning. The speed gain is gotten by merging nodes on discovery along with their history, and repeated history scanning for faster accelerations. All markings found are stored in a hash-using `Dictionary` which allows for fast adding of an unfound node, or adding to its history if it already exists.

5.2.6 Summary

The CovProcGraph algorithm [GRB10] shows promising results for some unbounded nets, but has abysmal performance on the bounded nets. It was the only algorithm besides the KarpMiller algorithm to hit the memory cap.

5.3 Results

The graphs show the amount of time taken, both in milliseconds and clock ticks, with about 3250 ticks per millisecond. The data was generated by running all combinations twenty times and averaging the results, with a timeout of one hour(timeout was 10 minutes in the *meshx2* nets). In case of Parallel versions, some outliers (probably due to delayed thread garbage collecting) were removed to get a better view of the time spent computing. Both the millisecond and tick axis are logarithmic, but the scale is different with each image, view with care.

In some cases, KarpMiller_Simple seems to perform better than KarpMiller_Speed/Parallel. This is because the execution quickly fails because of lack of memory.

Bounded nets: As expected, the performance on bounded nets is quite similar for MCT, ReySer and HanVal. They all simply explore the reachability graph of the net, while checking for accelerations. KarpMiller and CoverProc took way longer, as they have each marking in their data structure multiple times.

Unbounded nets: To our disappointment, it seems that most of the benchmarks used by the researched papers are hardly unbounded, that is: the transitions generate as much supply as they consume. The only reason these nets are unbounded is because some places are initially unbounded, and these ω tokens propagate

name	|P|	|T|	|MCS|	number of accelerations
lamport	11	9	14	bounded
newdekker	16	14	40	bounded
read-write	13	9	41	bounded
peterson	14	12	20	bounded
kanban	16	16	1	12
csm	14	13	16	3
fms	22	20	24	13
multipool	18	21	220	4
mesh2x2	32	32	256	12
mesh3x2	52	54	6400	20
pncsacover	31	36	80	11

Table 5.1: Benchmarks used.

throughout the net, even with "1:1 transitions". Only the net 'pncsacover' was *"truly unbounded"*: it starts without any ω tokens.

With these results, we can scrutinise the results of the analysed papers.

Geeraerts et al. compare their CovProc to the KarpMiller algorithm. [GRB10] Their results are questionable at best, as even on bounded nets CovProc would outperform KM, though the amount of Nodes and Pairs would suggest otherwise. The Petri net *RTP* (Figure B.3) only has markings consisting of a single supply. Yet constructing the KM tree of 16 nodes would take more time than constructing the 47 pairs in the CovProc algorithm. The net *kanban* (Figure B.4), is mistakenly shown as having a runtime longer than their timeout period of 20 minutes. The KM tree does not have 9839 nodes, as my implementation gives an OOM error at eight million nodes.

Not even the "Max Pairs" column of the CovProc algorithm is fair, as it seems to only count the amount of pairs of a single layer: the recursive calls made to construct an Oracle count their pairs separately, as shown by the kanban execution having a Max Pairs of just 4. This might also be why the runtime of the CovProc is so low: My execution of KM on the net *readwrite* (Figure B.1) took 15 milliseconds, not seven minutes. My execution of CovProc took 60 milliseconds, four times as much as the KM execution.

It is unknown whether this is a difference in measurement, or perhaps their implementation did not follow their specification as strictly. In kanban, there are 10 different transitions used for acceleration, and every different permutation of this order should lead to another recursive call, according to the line "$R_i = \bigcup_{m \in S} CovProc(\mathcal{N}, m)$ *where* $S \subseteq Flatten(\overline{Accel}(F_{i-1}))$". No matter how complete the first return value is, the other recursive calls are still made, in my implementation.

Reynier and Servais [RS13] copy these results to their own table, and justify this with *"Note that the implementation of* [7] *also was in Python, and the tests were run on the same computer."* If this is true, then the KM

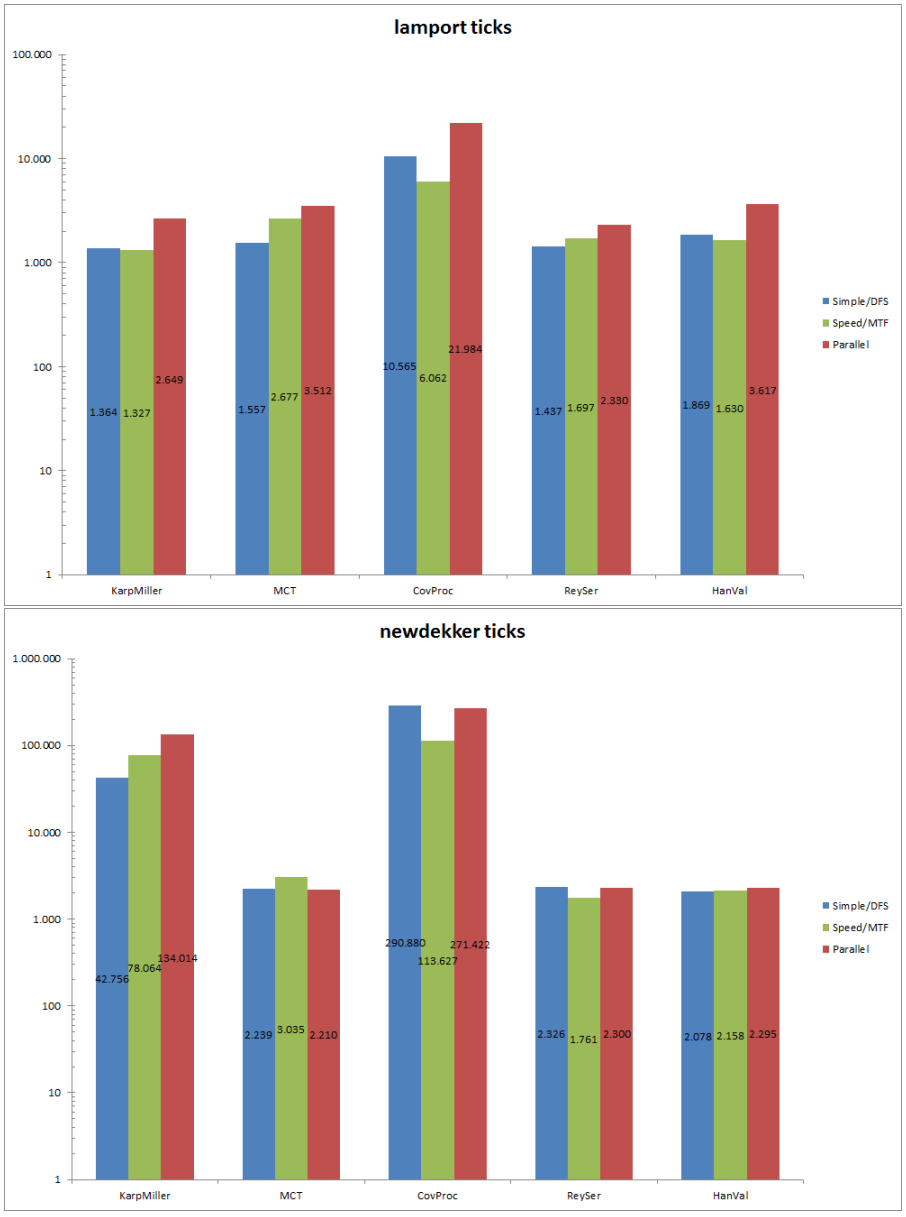


Figure 5.1: Execution results of lamport and newdekker.

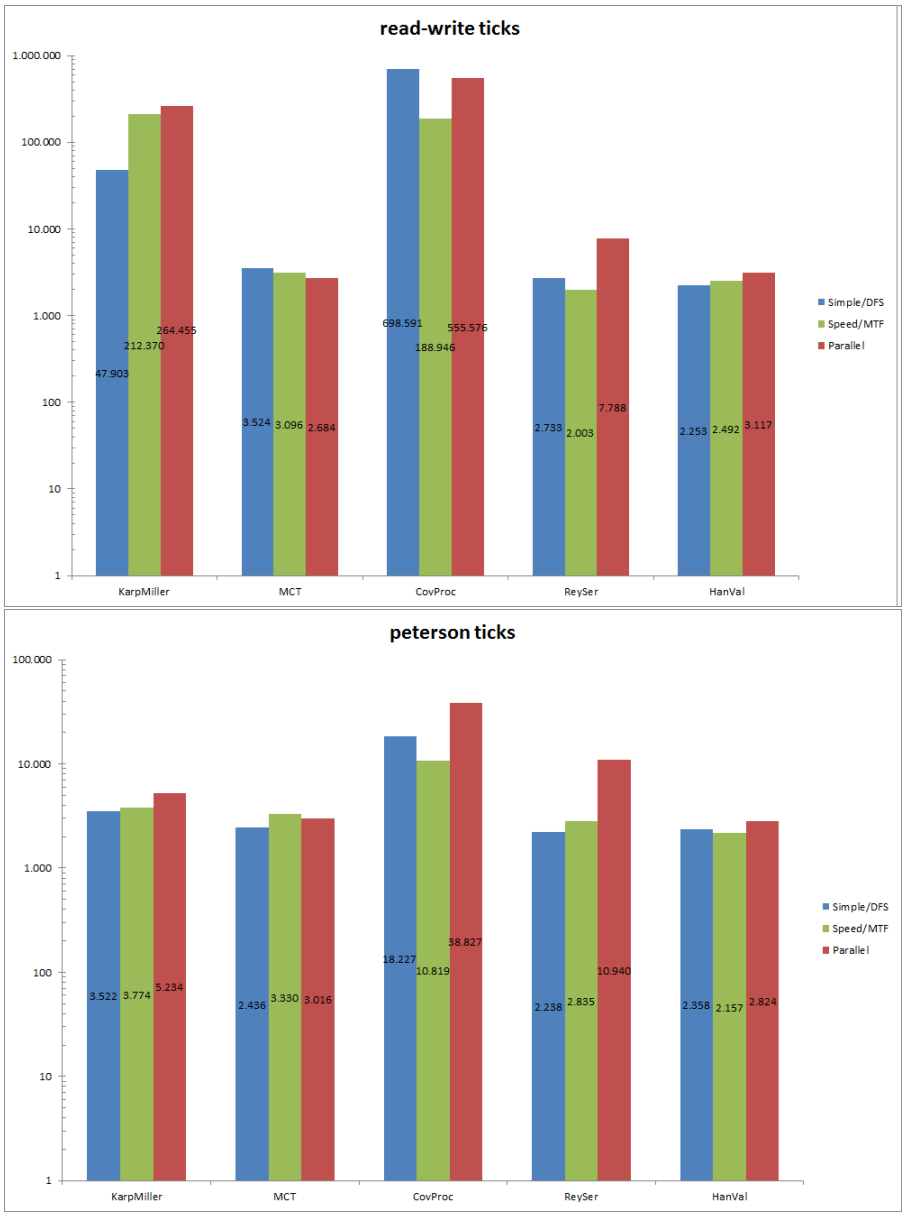


Figure 5.2: Execution results of read-write and peterson.

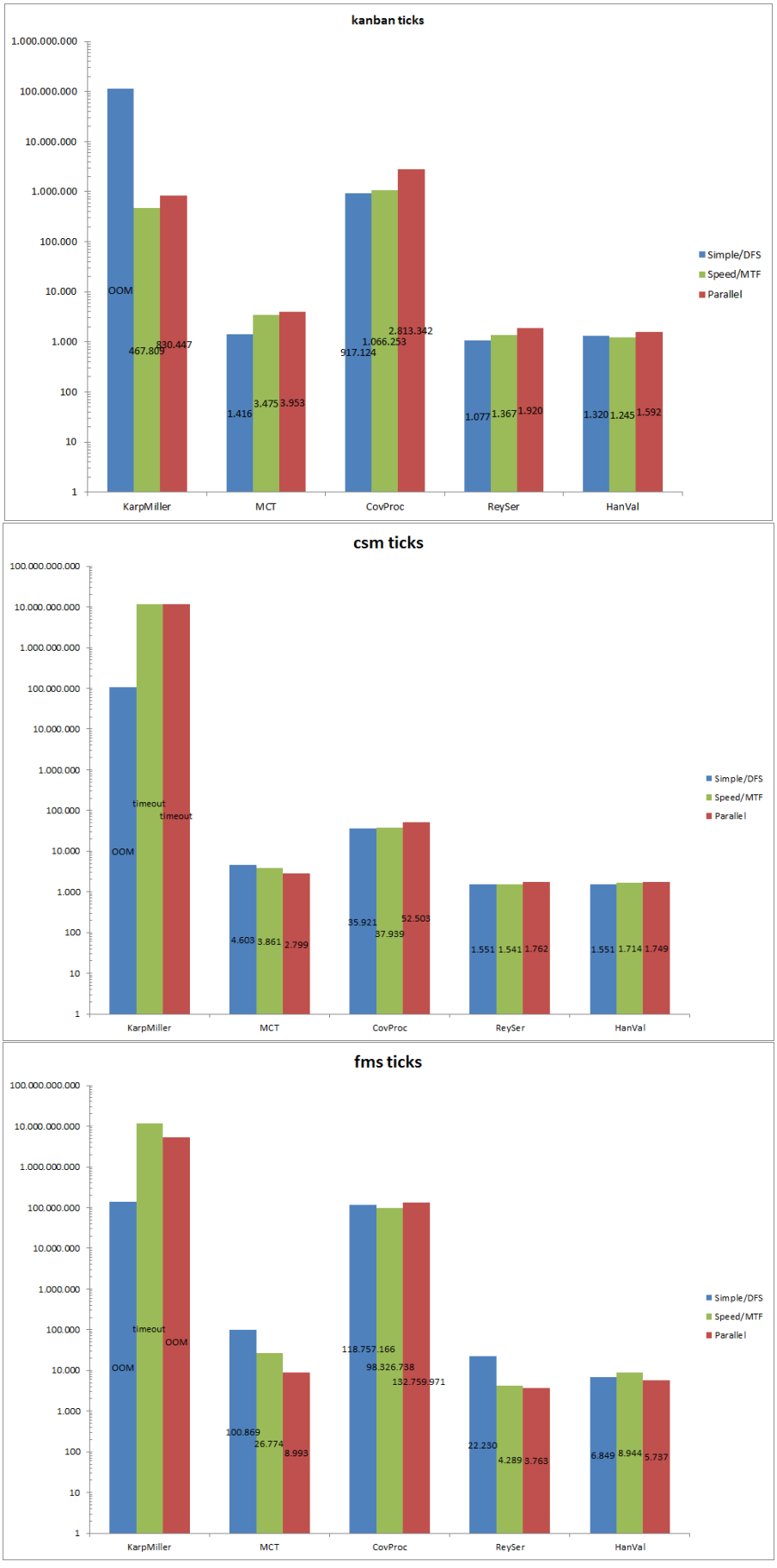


Figure 5.3: Execution results of kanban, csm and fms.

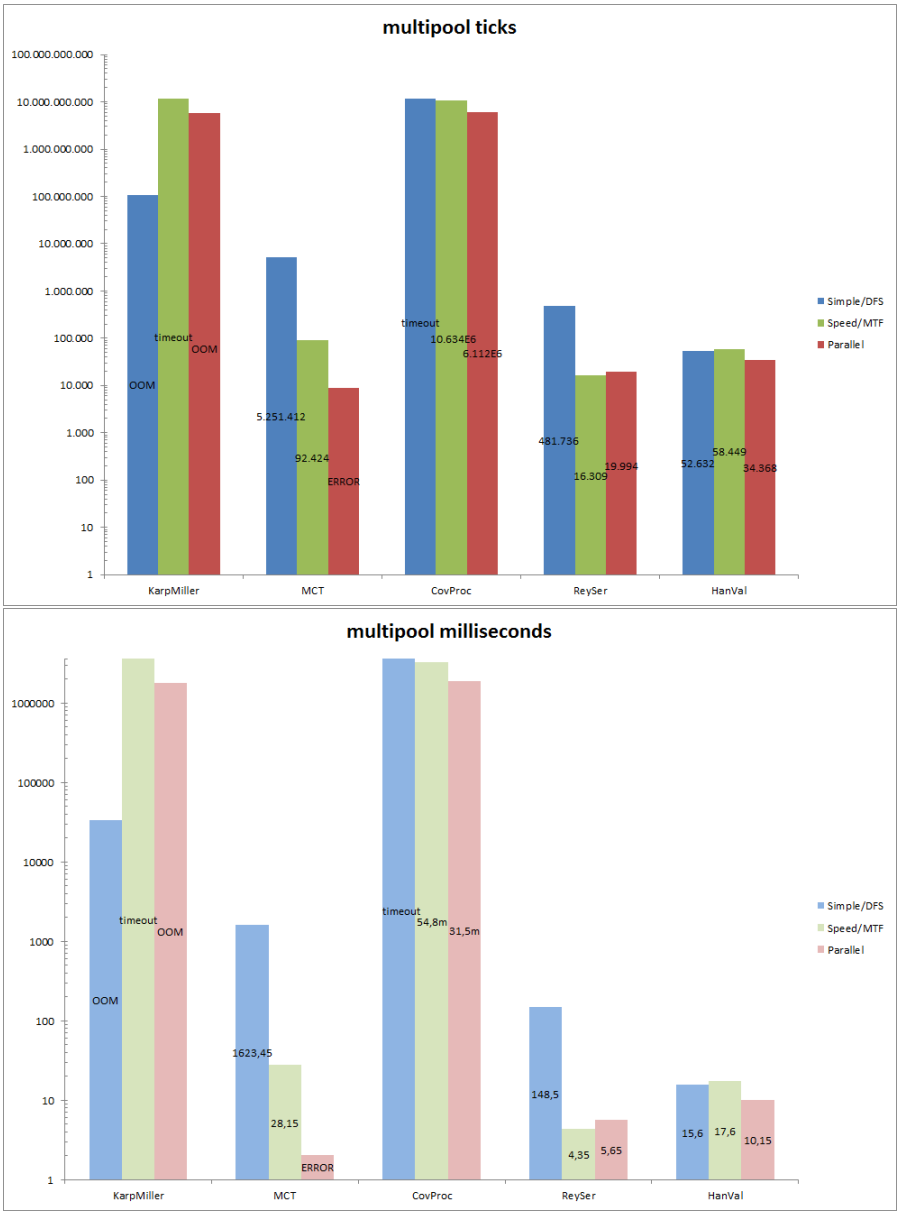


Figure 5.4: Execution results of multipool.

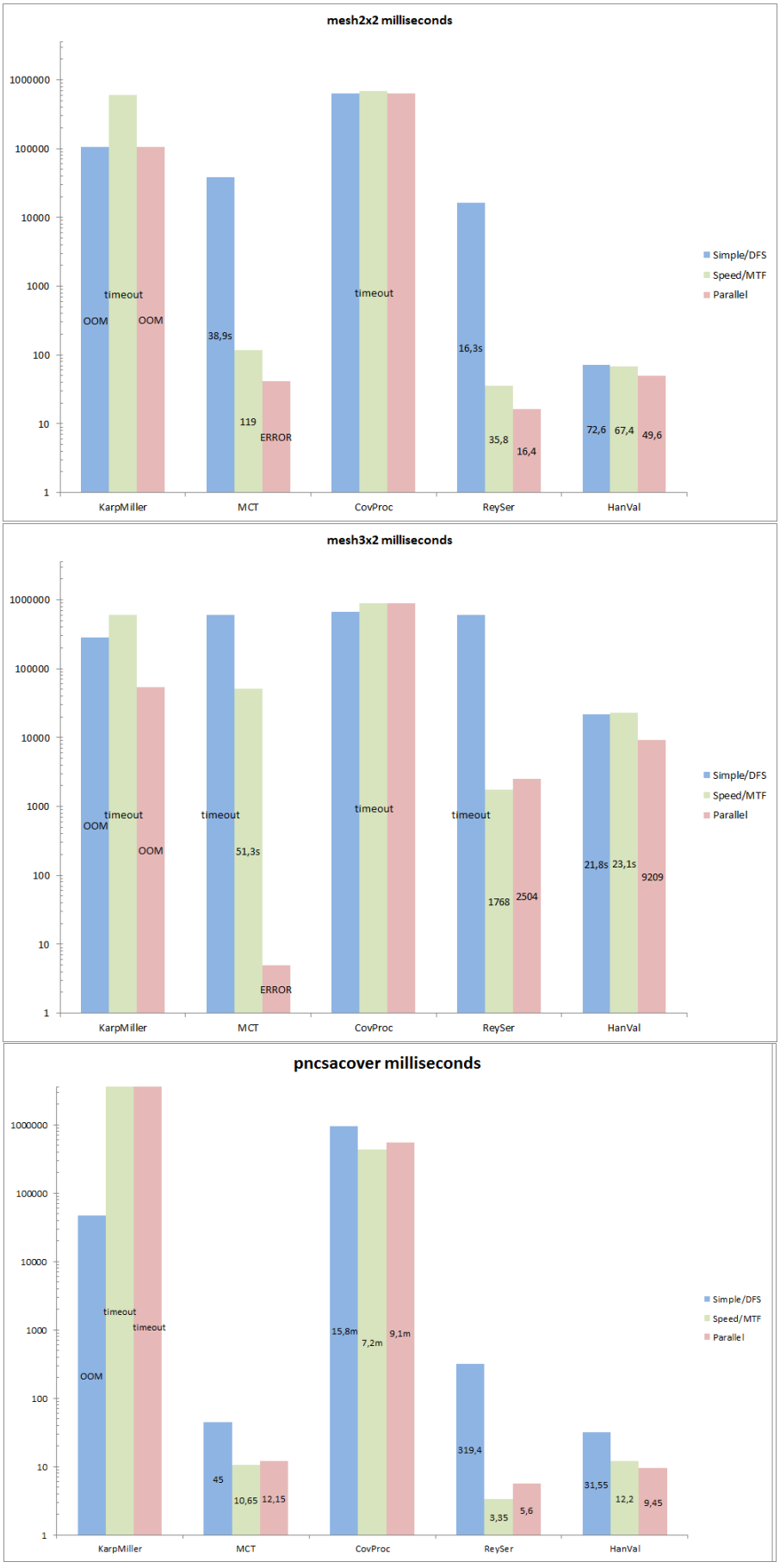


Figure 5.5: Execution results of mesh2x2, mesh3x2 and pncsacover.

implementation used here and in [GRB10] vary wildly, as their K&M execution of *readwrite* took 6.33 seconds, 60 times as fast. The *'#Wait'* column is the total number of nodes in the tree, except for the *kanban* net, which is said to have 72226 nodes with a runtime of just 9.1 seconds.

Nevertheless, the results for MP are different than mine as well. *mesh2x2* (Figure B.8) takes about as long (Simple is DFS), but multipool is done in 5ms instead of 5.2 seconds.

Lastly, Hansen and Valmari [VH12] do not compare runtimes, but *"the total numbers of constructed distinct ω-markings"*. But that is what this algorithm specializes in, at the expense of different checking. It compares this to the '#Wait' column of the previous paper, which is the *"number of elements passed in the waiting list"*. An element being a tuple (M, t), but in HanVal *"each ω-marking has an integer attribute next_tr containing a number of a transition"*, and deactivation *"is done simply by assigning to* [the removed marking].*next_tr a number that is greater than the number of any transition"*. HanVal has this number optimised but MP hasn't.

5.4 Summary

Earlier results are all based on the strong points of the newest algorithm, and comparisons are done against outdated and unoptimised results.

On bounded nets, there is hardly any difference. The MCT algorithm is just as effective, as its error is not encountered in bounded nets. MP and HanVal perform similarly; the same set of nodes is built, there are just different checks for acceleration and termination. Any differences in runtime are mere fractions of a second, and the overhead takes the longest amount of computation.

Space complexity is not an issue. Apart from KarpMiller and CovProc, the maximum memory used is often not even 10% above the size of the MCS.

What's also clear is that runtime is hardly ever an issue in the nets used. Only KarpMiller and CovProc struggle with some of them, both in time and space complexity. But the other algorithms usually run in under a millisecond, even the mesh3x2 net runs in under a minute. The clear winner is the MP algorithm in MTF order, as it is the fastest in all unbounded nets. Hansen and Valmari had a useful contribution to all the Speed versions with their MTF order. For any future comparisons to be useful, more complex Petri nets to test on will be needed.

Lastly, parallelisation is rarely a good idea, dependant on the net. Multithreading on smaller levels like checking ancestors might be more efficient, but is out of the scope of this thesis. Any subsequent accelerations while using MTF order basically means the worker threads are useless, only the workers that have the current maximum of ω tokens are being useful. If there are many accelerations like in kanban, parallelisation is merely extra overhead. It is useful for exploring the 'bounded component' of markings reachable from the same ω-marking M if M is in the MCS.

Chapter 6

Conclusions

We have reviewed several prominent papers in Chapter 3, containing algorithms for calculating the Minimal Coverability Set. The heuristics used vary between not pruning at all, to pruning a lot more than the Ur-algorithm MCG [Fin91]. Chapter 4 highlights some extra properties that can be derived from the results of the algorithms, like sequence detection. This can be useful when interested in answering specific problems as fast as possible, but does not matter if we are only interested in the MCS, in which case only memory consumption and runtime matter.

Analytical evaluation is not useful, as it is possible to construct specific nets so that any algorithm seems the superior one, as shown in the paper by Hansen and Valmari [VH12]. Only an evaluation of practical performance is unbiased, if the benchmarks used are a good representation of the intended use of the algorithms. Such an evaluation was done in Chapter 5. Earlier comparisons were also made in some papers, but the results were often years apart, computed by different computers, or coded in a different language. Implementations of the five algorithms were made in C#, and were run on several benchmarks, while tracking their memory consumption and runtime. Graphs of the runtimes are shown in Section 5.3.

However, the predominant set of benchmarks is no longer as useful for comparison, as runtimes approach mere milliseconds. Future testing of algorithms will mostly Memory is not an issue at all, as the set of elements during runtime is barely larger than the MCS. A runtime comparison shows the Karp&Miller algorithm and the CoverProc algorithms as slowest, often hitting the timeout cap. The best algorithm found was the Monotone-Pruning algorithm by Reynier and Servais, if it used the simple but effective Most-Tokens-First order devised by Hansen and Valmari. The effect of the order is huge, as the Depth-First order is a factor 100 slower, and the MTF version even beats the MCT algorithm in speed.

Appendix A

Program Code Sample

The full code is available from my supervisor and will be downloadable at the Leiden University website as well.

A.1 Program.cs

```
 using  System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Diagnostics; //Stopwatch


namespace PetriTester
{
    class Program
    {
        enum Status { Success, OutOfMemory, Canceled, Failure };
        static int TIMEOUTMS;

        static void Main(string[] args)
        {
            if (args.Length == 0 || args.Length > 3)
            {
                Console.WriteLine("Incorrect number of arguments.");
                Console.WriteLine("Usage: PetriTester <TestSize> [<timeoutms> [<Folder>]].");
                Console.WriteLine("Default Folder is current location.");
                Console.WriteLine("Default timeout is 10 minutes, use -1 to wait indefinitely.");
```

```
    return;
}
int TESTSIZE = int.Parse(args[0]);
TIMEOUTMS = (args.Length >= 2 ? int.Parse(args[1]) : 60000);
FileHelper.Folder = (args.Length == 3 ? args[2] : @".\");

PetriNet[] TheBenchmarks;
try
{
    TheBenchmarks = FileHelper.ReadBenchmarks();
}
catch (Exception e)
{
    Console.WriteLine("Error reading benchmarks in Benchmarks folder: {0}", e.Message);
    throw;
}

Type[] TheGraphers = new Type[] {
    typeof(KarpMiller_Simple),
    typeof(KarpMiller_Speed),
    typeof(KarpMiller_Parallel),

    typeof(MCT_Speed),
    typeof(MCT_Simple),
    typeof(MCT_Parallel),

    typeof(CovProcGraph_Simple),
    typeof(CovProcGraph_Speed),
    typeof(CovProcGraph_Parallel),

    typeof(ReySer_Simple),
    typeof(ReySer_Speed),
    typeof(ReySer_Parallel),

    typeof(HanVal_DFS),
    typeof(HanVal_MTF),
    typeof(HanVal_Parallel),
};

SortedDictionary<string, List<Result>> results = FileHelper.LoadData();
if (results == null)
{
```

```
        Console.WriteLine("No previous valid results.dat found.");
        results = new SortedDictionary<string, List<Result>>();
    }
    else
    {
        Console.WriteLine("Found results.dat, rewriting it to be clearer...");
        FileHelper.SaveData(results);
    }

    CancellationTokenSource cts = new CancellationTokenSource();
    Dictionary<string, int> failedRuns = new Dictionary<string, int>();

    for (int test = 0; test < TESTSIZE; test++)
    {
        foreach (PetriNet pn in TheBenchmarks.Randomized())
        {
            foreach (Type grapherType in TheGraphers.Randomized())
            {
                string key = pn.ToString() + ", " + grapherType.ToString().Split('.').Last();
                List<Result> list;
                if (results.TryGetValue(key, out list) && list.Count >= TESTSIZE)
                    continue;

                GC.Collect();
                GC.WaitForPendingFinalizers();

                Grapher grapher = (Grapher)Activator.CreateInstance(grapherType);
                grapher.Init(pn);

                Status status; //for debugging
                cts = new CancellationTokenSource(TIMEOUTMS);
                cts.Token.Register(MessageAborting, false);
                Stopwatch sw = Stopwatch.StartNew();

                try
                {
                    Console.Write("Executing benchmark {0} with grapher {1}...", pn.ToString(),
                        grapher.ToString());
                    grapher.Run(cts.Token);
                    status = Status.Success;
                    Console.WriteLine("Execution of benchmark {0} with grapher {1} succeeded. ",
                        pn.ToString(), grapher.ToString());
```

```
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Execution of benchmark {0} with grapher {1} timed out. ",
                    pn.ToString(), grapher.ToString());
                status = Status.Canceled;
            }
            catch (OutOfMemoryException)
            {
                Console.WriteLine("Out of memory executing benchmark {0} with grapher {1}.
                    ", pn.ToString(), grapher.ToString());
                status = Status.OutOfMemory;
            }
            #if !DEBUG
            catch (Exception)
            {
                Console.WriteLine("Execution of benchmark {0} with grapher {1} FAILED.",
                    pn.ToString(), grapher.ToString());
                status = Status.Failure;
                if (failedRuns.ContainsKey(key))
                    failedRuns[key]++;
                else
                    failedRuns.Add(key, 1);
            }
            #endif
            sw.Stop();
            cts.Dispose();

            bool tryAgain = true;
            Result result;
        saveresult:
            Console.Write("Creating Result...");
            try
            {
                result = new Result(sw, grapher);
                if (list != null)
                {
                    list.Add(result);
                }
                else
                {
                    results.Add(key, new List<Result> { result });
```

```
                    }
                    FileHelper.SaveResult(key, result);
                }
                catch
                {
                    if (tryAgain)
                    {
                        tryAgain = false;
                        Console.Write("Error while creating Result, retrying...");
                        GC.Collect();
                        goto saveresult;
                    }
                    Console.Write("Error while creating Result, giving up...");
                    return;
                }
                Console.WriteLine("Appended result to file.");
                #if DEBUG
                Console.WriteLine(result.Markings);
                #endif
            }//foreach grapher
        }//foreach benchmark
    }//for testsize

    if (failedRuns.Count == 0)
    {
        Console.WriteLine("No failed runs. Rewriting save data to be clearer..");
        FileHelper.SaveData(results);
    }
    else
    {
        Console.WriteLine("Failed runs:");
        foreach (var kvp in failedRuns)
            Console.WriteLine("{0} failed {1} time(s).", kvp.Key, kvp.Value);
    }
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}//Main

static void MessageAborting()
{
    Console.Write("Execution took longer than {0}, now aborting...", TIMEOUTMS);
}
```

```
    }//Program

}//namespace
```

A.2 KarpMiller.cs

```
 using  System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace PetriTester
{
    class KarpMiller_Speed : Grapher
    {
        //Karp and Miller did not define a procedure to generate the tree, as it was more about the
            mathematics than the computation.
        //Instead, "Let \eta(PN) be a tree associated with a Petri Net PN such that <some conditions>
            hold."

        HashSet<Marking> nodes;

        protected override IEnumerable<Marking> GetFinalMarkings()
        {
            var rv = nodes.Minimize();
            nodes.Clear();
            return rv;
        }

        public override void Run(CancellationToken CancellationToken)
        {
            //This is a faster implementation of the Karp&Miller algorithm, which focuses on
                calculating the MCS as fast as possible, other data is redundant.
            //As there is no pruning whatsoever, we can construct the tree in whichever order we want.
            //We have chosen for a DFS approach, as it is easier to track ancestors of the current
                node that way.
            //Instead of following parent pointers, there is simply a set of ancestors. Nodes are
                added when explored, and removed when backtracked from.
```

```
//The nodes currently being explored can be put in a Stack, where backtracking is done by
    popping from the stack.
//We still explore Markings we have already explored before, or the algorithm would change
    too much.
//The amount of Memory taken is cut back significantly, the total runtime not at all.
Stack<int> transitionCount = new Stack<int>();
Stack<Marking> unfinishedMarkings = new Stack<Marking>();

HashSet<Marking> ancestors = new HashSet<Marking>(Marking.EqualityComparer);
nodes = new HashSet<Marking>();

Marking currentMarking = initialMarking;

ancestors.Add(currentMarking);
nodes.Add(currentMarking);
int currentTransition = 0;

while (true)//unfinishedMarkings.Count != 0)
{
    CancellationToken.ThrowIfCancellationRequested();
    if (currentTransition == transitions.Length)
    {
        //this marking is done

        //if completely done
        if (unfinishedMarkings.Count == 0)
            return;

        //else we move on to a sibling, so currentNode is not an ancestor anymore
        ancestors.Remove(currentMarking);

        currentTransition = transitionCount.Pop();
        currentMarking = unfinishedMarkings.Pop();
    }
    else if (currentMarking.IsFireable(transitions[currentTransition]))
    {
        Marking newMarking = currentMarking.Fire(transitions[currentTransition]);

        //pump marking
        Marking pumpedMarking = new Marking(newMarking);

        foreach (Marking ancestor in ancestors)
```

```
                {
                    if (ancestor < newMarking)
                        pumpedMarking.Pump(ancestor);
                }

                if (ancestors.Add(pumpedMarking))
                {
                    CurrentMemory++;

                    //explore current node later
                    unfinishedMarkings.Push(currentMarking);
                    transitionCount.Push(currentTransition + 1);

                    nodes.Add(pumpedMarking);

                    currentMarking = pumpedMarking;
                    currentTransition = 0;
                }
                else
                    currentTransition++;
            }//if fireable
            else
                currentTransition++;
        }//while true

    }//Run

}//KarpMillerGraph

}
```

A.3 MCT.cs

```
 using  System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace PetriTester
{
    //as described on page 229 in
    //THE MINIMAL COVERABILITY GRAPH FOR PETRI NETS
    //by Alain FINKEL

    //This version will focus on generating the MCS. Arcs are deemed irrelevant,
    //and instead of two separate procedures MCT and MCG, nodes are identified with each other as
        soon as discovered.
    class MCT_Speed : Grapher
    {
        //Version of MCS that is focused on producing the MCS as quickly as possible.
        //It therefore has no Arc objects that store transitions,
        //and Nodes are a subtype of Marking, so that Nodes are identified by their Marking.
        //processedNodes and unprocessedNodes are now MTFHashSets that prioritize markings with a
            large number of Omega tokens.

        private MTFHashSet processedNodes;
        private MTFHashSet unprocessedNodes;

        protected override IEnumerable<Marking> GetFinalMarkings()
        {
            return processedNodes;
        }

        public class Node : Marking
        {
            public bool removed = false;
            public Node parent = null;
            public List<Node> children;

            public Node(Marking m) : base(m)
            {
                children = new List<Node>();
            }
```

```
    public Node(Marking m, Node parent) : base(m)
    {
        this.parent = parent;
        children = new List<Node>();
    }
    public new Node Fire(Transition t)
    {
        return new Node(supply.Add(t.effect));
    }
}

public override void Run(CancellationToken CancellationToken)
{
    #region original description
    /*
     * procedure minimal_coverability_graph(PN: Petri net; vat MCS: set of markings; vat MCG:
         graph);
    {* the result will be in MCG * }
    begin
        minimal_coverability_tree(PN; MCS; MCT);
        identify_nodes_having_same_label(MCT; MCG);
        {* the procedure "identify_nodes_having_same_label(T: tree; G: graph)" transforms the
            tree T into a graph G such that two nodes in T having the same label are
            identified in G * }
        for every arc (m,t,m') of MCG do
        {* after having identify nodes with the same label, we confuse without ambiguity a
            node and its label * }
        if not ( m(t>m' ) then remove_arc((m,t,n'); MCG);
        {* the procedure "romove_arc((m,t,m'); ear G))" only removes the arc (m,t,m') from G *}
    end;
     *
    procedure minimal_coverability_tree(PN: Petri net; var MCS: set of markings; var MCT:
        tree);
    {* the result will be in MCT *}
        var unprocessednodes, processednodes: set of nodes; n, n', n1, n2: node; t:
            transition; ancestor: boolean;
        begin
            unprocessednodes := { create node(r,M_0) }; {* M_0 is the marking of root r *}
            processednodes := \emptyset; {* processednodes will be the minimal coverability set
                *}
            while unprocessednodes \neq \emptyset do
            begin
```

```
select some node n \in unprocessednodes;
unprocessednodes := unprocessednodes - {n};
case n : [1..4] of {* m is the marking of n and m_1 is the marking of n_1 * }
1: there is a node n_1 \in processednodes such that m = m_1 :
begin
    processednodes := processednodes + {n};
    exit; {* exit of case *}
    end;
* 2: there is a node n_1 \in processednodes such that m < m_1 :
begin
    remove_node(n; MCT);
    exit; {* exit of case * }
    end;
{* the procedure "remove__node(n: node; var T: tree)" removes the node n and
     the arc from the
direct ancestor of n to n, in the tree T * }
3: there is a node n_1 \in processednodes such that m_1 < m :
begin
    m_2 := m; ancestor := false;
    for all ancestors n_1 of n such that m_1 < m do
        for all places p such that m_1(p) < m(p) do m_2(p) := \omega;
    if there is an ancestor n_1 of n such that m_1 < m_2 then
    begin
        ancestor := true;
        n_1 := first node processed, on the path from the root to n such that
            m_1 < m_2;
        m_1 := m_2;
        remove_tree(n_1; MCT);
        {* the procedure "remove_tree(n: node; var T: tree)" which removes the
            subtree
        whose root is n in the tree T (note that we keep the root n) * }
        remove from(processednodes+unprocessednodes) all nodes of tree(n_1 ;MCT);
        unprocessednodes := unprocessednodes + {n_1};
        end;
    else
        unprocessednodes := unprocessednodes + {n};
    for every n_1 \in processednodes such that m_1 < m_2 do
    begin
        remove from(processednodes+unprocessednodes) all nodes of tree(n_1;MCT;
        remove_tree(n_1; MCT;
        remove_node(n_1; MCT);
        end;
```

```
                exit; {* exit of case * }
                end;
            4: otherwise :
            begin
                for every transition t such that m(t>m' do
                begin
                    create_node+arc((n,t,n'); MCT; { * m' is the marking of the new node n'
                        * }
                    {* the procedure "create_node+arc((n,t,n'); T)" creates a new node n'
                        labelled by m'
                    and a new arc (n,t,n') in the tree T * }
                    unprocessednodes := unprocessednodes + {n'};
                    end;
                processednodes := processednodes + {n};
                exit; { * exit of case * }
                end;
            end; {* end of case *}
            unprocessednodes := maximal(unprocessednodes);
            {* the function "maximal(S : set) : set" computes the set of nodes n such that
                every label(n) is maximal *}
            MCS := {label(n) ; n \in processednodes };
            end; {* end of while *}
        end; { * end of procedure *}
 */
#endregion original description

#region optimisations
/*
 * 1. Keeping unprocessedNodes maximized every iteration only needs to check the new
     elements.
 * 2. Keeping multiple nodes with the same marking is useless, it would only get removed
     in MCG
 * 3. The node pushed in case 3 has a label that was just pumped: there is no comparable
     element in processednodes.
 *  Just fall through to case 4 with this new node instead of pushing and popping it.
 * 4. As the assumption is that the order shouldn't matter for correctness(we know it does
     though),
 *  use a Most-Tokens-First set to designate the next node to treat.
*/
#endregion

processedNodes = new MTFHashSet(numPlaces, Marking.EqualityComparer);
```

```
unprocessedNodes = new MTFHashSet(numPlaces, Marking.EqualityComparer);

Node currentNode, newNode, firstAncestor;
List<Node> list;

unprocessedNodes.Add(new Node(initialMarking, null));
//parent.Add(initialMarking, null);

while (unprocessedNodes.Count != 0)
{
    CancellationToken.ThrowIfCancellationRequested();

    currentNode = (Node)unprocessedNodes.Pop();

    //case 1 is obsolete, no node is in the tree twice

    //case 2
    if (processedNodes.NoSmallerThan(currentNode).Any(LargerThan(currentNode)))
    {
        //remove the node by removing the 'arc' to it
        currentNode.parent.children.Remove(currentNode);
        continue;
    }

    //case 3
    //check if any ancestors are smaller
    Node pumpedNode = new Node(currentNode, null);
    firstAncestor = null;
    for (Node walker = currentNode.parent; walker != null; walker = walker.parent)
        if (walker < currentNode)
        {
            firstAncestor = walker;
            pumpedNode.Pump(walker);
        }
        else if (walker < pumpedNode)
            firstAncestor = walker;

    //if a pump has happened
    if (firstAncestor != null)
    {
        processedNodes.Remove(firstAncestor);
```

```
    //replace first smaller Ancestor by newNode
    if (firstAncestor.parent != null)
    {
        //make parent point to newNode instead
        list = firstAncestor.parent.children;
        list.Remove(firstAncestor);
        list.Add(pumpedNode);
        //remove obsolete entries
        pumpedNode.parent = firstAncestor.parent;
    }

    RemoveTree(firstAncestor);

    currentNode = pumpedNode;
}
Node[] smallers =
    processedNodes.NoLargerThan(currentNode).Where(SmallerThan(currentNode)).Cast<Node>().ToArray();
foreach (var smaller in smallers)
{
    if (smaller.removed == false)
    {
        smaller.removed = true;
        processedNodes.Remove(smaller);
        //smaller always have a parent, root would be removed as firstAncestor
        smaller.parent.children.Remove(smaller);
        RemoveTree(smaller);
    }
}

//case 4
list = currentNode.children;
foreach (Transition t in transitions)
    if (currentNode.IsFireable(t))
    {
        newNode = currentNode.Fire(t);
        //don't add duplicate nodes
        if (!processedNodes.Contains(newNode) && !unprocessedNodes.Contains(newNode))
        {
            newNode.parent = currentNode;
            list.Add(newNode);
        }
    }
```

```
        foreach (Marking m in list)
            if (!unprocessedNodes.NoSmallerThan(m).Any(LargerThan(m)))
            {
                unprocessedNodes.Add(m);
                //RemoveSmaller(m);
                foreach (Marking rm in
                    unprocessedNodes.NoLargerThan(m).Where(SmallerThan(m)).ToArray())
                    unprocessedNodes.Remove(rm);
            }

        processedNodes.Add(currentNode);
        CurrentMemory = processedNodes.Count + unprocessedNodes.Count;
    }//while unprocessednodes

}//Run


/// <summary>
/// Removes the descendants of root from processedNodes and unprocessedNodes.
/// </summary>
/// <param name="removeMe"></param>
private void RemoveTree(Node root)
{
    Stack<Node> removeThese = new Stack<Node>(root.children);
    Node node;

    while(removeThese.Count != 0)
    {
        node = removeThese.Pop();
        if (processedNodes.Remove(node))
        {
            //remove its children too
            foreach (Node n in node.children)
                removeThese.Push(n);
            node.children.Clear();
        }
        else
            unprocessedNodes.Remove(node);
        node.parent = null;
    }
```

```
            root.children.Clear();
        }

        Func<Marking, bool> LargerThan(Marking marking)
        {
            return (Marking m) => m > marking;
        }
        Func<Marking, bool> SmallerThan(Marking marking)
        {
            return (Marking m) => m < marking;
        }
    }//KarpMillerGraph

}
```

A.4 CovProc.cs

```
 using  System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace PetriTester
{
    //as described in
    //ON THE EFFICIENT COMPUTATION OF THE MINIMAL
    //COVERABILITY SET OF PETRI NETS
    //by GILLES GEERAERTS, JEAN-FRANCOIS RASKIN, LAURENT VAN BEGIN
    //International Journal of Foundations of Computer Science(Oct 5, 2009)
    class CovProcGraph_Speed : Grapher
    {

        HashSet<Pair> oracle;
        HashSet<Pair> visited;
        static CancellationToken CancellationToken;

        class Pair : IEquatable<Pair>
        {
            public Marking m1;
            public Marking m2;
```

```
bool hashed = false;
int hash;

public Pair(Marking m1, Marking m2)
{
    this.m1 = m1;
    this.m2 = m2;
}

public bool SquareSubset(Pair pair2)
{
    return pair2.m1 >= this.m1 && pair2.m2 >= this.m2
        && pair2.m2 - pair2.m1 >= this.m2 - this.m1;
    //pairs mean "you can get from m1 to m2", so m2 will have omega tokens at least at the
        same places as m1.
    //the minus here is well defined then
}
/// <summary>
/// Checks equality, first by reference, then by HashCode, then by contents.
/// </summary>
/// <param name="other"></param>
/// <returns></returns>
public bool Equals(Pair other)
{
    if (ReferenceEquals(this, other))
        return true;
    return GetHashCode() == other.GetHashCode()
        && ReferenceEquals(this.m1, other.m1) || m1.EqualContents(other.m1)
        && ReferenceEquals(this.m2, other.m2) || m2.EqualContents(other.m2);
}
public override string ToString()
{
    return string.Format("[{0}, {1}]", m1 ,m2);
}
public override int GetHashCode()
{
    if (!hashed)
    {
        unchecked
        {
            hash = (m1.GetHashCode() << 16) + (m1.GetHashCode() >> 16)
                    + m2.GetHashCode();
```

```
            }
            hashed = true;
        }
        return hash;
    }
}

public CovProcGraph_Speed() { }
CovProcGraph_Speed(CovProcGraph_Speed cpg, Marking marking)
{
    numPlaces = cpg.numPlaces;
    transitions = cpg.transitions;
    initialMarking = marking;
}

override protected IEnumerable<Marking> GetFinalMarkings()
{
    return Flatten(oracle.Concat(visited)).Minimize();
}

public override void Run(CancellationToken CancellationToken)
{
    CovProcGraph_Speed.CancellationToken = CancellationToken;
    RunInternal();
}

void RunInternal()
{
    #if DEBUG
    int iteration = 0;
    #endif

    //They're all HashSets so that checks for duplicates can quickly be made. See the
        MaxSquare functions.
    oracle = new HashSet<Pair>();
    visited = new HashSet<Pair>();
    HashSet<Pair> frontier = new HashSet<Pair>();
    HashSet<Pair> recursive = new HashSet<Pair>();
    frontier.Add(new Pair(initialMarking, initialMarking));
    Marking[] iterationCondition1;
    Marking[] iterationCondition2 = new Marking[0];
    bool oracleChanged = false;
```

```
do
{
    CancellationToken.ThrowIfCancellationRequested();

    #if DEBUG
    iteration++;
    #endif
    iterationCondition1 = iterationCondition2;
    foreach (Marking marking in Flatten(Accel(frontier)))
    {
        CovProcGraph_Speed cpg = new CovProcGraph_Speed(this, marking);
        try
        {
            cpg.RunInternal();
        }
        finally
        {
            //record memory even when timed out
            long temp = cpg.maxMemory;
            CurrentMemory += temp;
            CurrentMemory -= temp;
        }
        recursive = MaxSquarev2(recursive, Minimize(cpg.oracle.Concat(cpg.visited)));
    }
    if (recursive.Count != 0)
    {
        oracle = MaxSquarev2(recursive, oracle);
        oracleChanged = true;
        recursive.Clear();
    }

    //V_i:=  Max  ( V_i1       F_i1 ) \       (O_i);
    //frontier_i-1 has already had its elements inferior to visited_i-1 removed
    //frontier_i-1's elements are already relevant wrt. visited_i-1
    //we need to check if visited's elements are still relevant wrt. frontier
    visited.RemoveWhere(pair1 =>
        frontier.Any(pair2 => pair1.SquareSubset(pair2)));
    visited.UnionWith(frontier);
    //if oracle hasn't changed, then both frontier_i-1 and visited_i-1 are already maximal
        wrt oracle, no need to check again
    if (oracleChanged)
```

```
            visited.RemoveWhere(pair1 =>
                oracle.Any(pair2 => pair1.SquareSubset(pair2)));

        //F_i:=  Max  (Post( F_i1 )     Accel( F_i1 )) \       (O_i    V_i);
        frontier = Minimize(Post(frontier)
                            .Concat(Accel(frontier)));

        if (oracleChanged)
            frontier.RemoveWhere(pair1 =>
                oracle.Any(pair2 => pair1.SquareSubset(pair2)));
        frontier.RemoveWhere(pair1 =>
            visited.Any(pair2 => pair1.SquareSubset(pair2)));

        CurrentMemory = oracle.Count + visited.Count + frontier.Count;

        oracleChanged = false;
        iterationCondition2 = Flatten(oracle.Concat(visited)).ToArray();
    } while (iterationCondition2.Any(m2 =>     //while there exists an object in flatten_i
        !iterationCondition1.Any(m1 => m1 >= m2))); //not covered by any object in flatten_i-1
}//Run

/// <summary>
/// Generates a List of new Pairs, based on firing transitions from the pairs in set. Does not
    remove duplicates.
/// </summary>
/// <param name="set"></param>
/// <returns></returns>
List<Pair> Post(IEnumerable<Pair> set)
{
    List<Pair> rv = new List<Pair>();
    foreach (Pair pair in set)
    {
        rv.AddRange(Post(pair));
    }
    return rv;
}
IEnumerable<Pair> Post(Pair pair)
{
    foreach (Marking m in Post(pair.m2))
    {
        yield return new Pair(pair.m1, m);
        yield return new Pair(pair.m2, m);
```

```
        }
    }
    IEnumerable<Marking> Post(Marking m)
    {
        foreach (Transition t in transitions)
        {
            if (m.IsFireable(t))
                yield return m.Fire(t);
        }
    }

    /// <summary>
    /// Generates a List of new Pairs, based on possible accelerations within set. Does not remove
        duplicates.
    /// </summary>
    /// <param name="set"></param>
    /// <returns></returns>
    List<Pair> Accel(IEnumerable<Pair> set)
    {
        List<Pair> rv = new List<Pair>();
        foreach (Pair pair in set)
        {
            if (pair.m2 > pair.m1)
                rv.Add(new Pair(pair.m2, AccelPair(pair)));
        }
        return rv;
    }
    Marking AccelPair(Pair pair)
    {
        Marking rv = new Marking(pair.m2);
        rv.Pump(pair.m1);
        return rv;
    }

    /// <summary>
    /// Returns only the second elements of the pairs, removing duplicate markings.
    /// </summary>
    /// <param name="set"></param>
    /// <returns></returns>
    IEnumerable<Marking> Flatten(IEnumerable<Pair> set)
    {
        return set.Select(pair => pair.m2).Distinct(Marking.EqualityComparer);
```

```
}

HashSet<Pair> Minimize(IEnumerable<Pair> input)
{
    return input.Aggregate(new HashSet<Pair>(), (set, pair1) =>
    {
        if (set.RemoveWhere(pair2 => !ReferenceEquals(pair1, pair2)
                                && pair2.SquareSubset(pair1)) > 0
            || set.All(pair2 => ReferenceEquals(pair1, pair2)
                            || !pair1.SquareSubset(pair2)))
            set.Add(pair1);
        return set;
    });
}

/// <summary>
/// Returns the maximal elements(wrt the SquareSubset relation) of two already maximized sets.
/// </summary>
/// <param name="set1"></param>
/// <param name="set2"></param>
/// <returns></returns>
HashSet<Pair> MaxSquarev2(HashSet<Pair> set1, HashSet<Pair> set2)
{
    if (set1.Count == 0)
        return set2;
    //Pairs in both sets are maximal in both sets, thus maximal in the union.
    //Those elements can be returned, and no other elements need to be compared to them.
    HashSet<Pair> rv = new HashSet<Pair>(set1.Intersect(set2));
    set1.ExceptWith(rv);
    set2.ExceptWith(rv);
    //pairs can be approved or denied by a SquareSubset check
    //if in one set, then not the other, as p1 <= p2 <= p1' can not happen in maximized sets
    //all pairs are unique, the intersection is already filtered
    HashSet<Pair> approved = new HashSet<Pair>();
    HashSet<Pair> denied = new HashSet<Pair>();

    foreach (Pair pair1 in set1)
    {
        foreach (Pair pair2 in set2)
        {
            //check #1
            if (!approved.Contains(pair1) && pair1.SquareSubset(pair2))
```

```
                {
                    //pair1 is denied, pair 2 is approved
                    denied.Add(pair1);
                    approved.Add(pair2);
                    break; //no further checking is needed for this pair1
                }
                //check #2
                if (!approved.Contains(pair2) && pair2.SquareSubset(pair1))
                {
                    approved.Add(pair1);  //keep checking, pair1 might be the single pair that
                        invalidates some pair2
                    denied.Add(pair2);
                }
            }//for pair2 in set2
            if (!denied.Contains(pair1))
                rv.Add(pair1);

            //don't check denied elements in further iterations
            set2.ExceptWith(denied);
            denied.Clear();
        }//for pair1 in newSet1

        //whatever wasn't denied, and is thus still in set2, is a maximal element
        rv.UnionWith(set2);
        return rv;
    }

}//KarpMillerGraph

}
```

A.5 ReySer.cs

```
 using  System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace PetriTester
{
    //Faster version of the Monotone Pruning Algorithm.
```

```
//Uses a Most-Tokens-First approach to explore the set of unfinished nodes.

//Does not keep track of transitions, does not even remember the whole tree,
//What is remembered is a tree of active nodes, for each node, possibly a list of inactive nodes
    associated with it.
//Optimized detection of covered nodes in lines 10/11.
class ReySer_Speed : Grapher
{
    HashSet<Node> Act;

    override protected IEnumerable<Marking> GetFinalMarkings()
    {
        return Act;
    }

    override public void Run(CancellationToken CancellationToken)
    {
        #region Original Description
        //the Monotone Pruning Algorithm
        //Algorithm 2 Monotone Pruning Algorithm for Petri Nets.
        //Require: A Petri net N = (P, T, I, O, m0).
        //Ensure: A labelled tree C = (X, x0, B, \Lambda) and a set Act  X such that \Lambda(Act)
            = MCS(N).
        //1: Let x0 be a new node such that \Lambda(x0) = m0;
        //2: X:= {x0}; Act:= X; Wait:= {(x0, t) | Lambda(x0) \fire{t} }; B:= ;
        //3: while Wait \neq  do
        //4:   Pop( n , t) from Wait.
        //5:   if  n      Act then
        //6:       m:= Post(Lambda( n ), t);
        //7:       Let n be a new node such that Lambda(n) = Acc(Lambda(Ancestor_C(n)  Act),m);
        //8:       X:= X    {n}; B:= B    {( n , t, n)};
        //9:       if Lambda(n) is not covered by anything in Lambda(Act) then
        //10:          Act:= Act \ { x | y     Ancestor_C(x) s.t. Lambda(y)  Lambda(n)   (y    Act
               y \not\in Ancestor_C(n))};
        //11:          Act:= Act    {n}; Wait:= Wait   {(n, u) | \Lambda(n) \fire{u} };
        //12:      end if
        //13:  end if
        //14: end while
        //15: Return C = (X, x0, B, \Lambda) and Act.

        // p      P, Acc(M, m)(p) =:  if    m       M |  m  < m     m (p) < m(p) <  ,
        //                   =: m(p) otherwise.
```

```
#endregion

#region Optimizations
//**Original optimizations**
//In order to minimize the overhead of the inactive nodes, our implementation takes
    advantage of the following observation.
//One of the following two cases occurs when a node x, different from the new node n, is
    deactivated:
//    if the new node n is not a descendant of x, then x and its subtree are completely
    removed
//    (indeed the algorithm will not need them anymore);
//    if the new node n is a descendant of x an acceleration has occurred between n and an
    ancestor y of x;
//    the node x could be used later to deactivate one of the descendants of node n;

//in our implementation x
//    is removed and the set of markings of the deactivated nodes lying between y and n
    (including x)
//    is associated with n; nevertheless, note that we only need to keep the minimal
    elements of this set.

//Therefore our implementation maintains a tree data structure containing only the active
    nodes and for
//each active node n it maintains the set of the minimal markings of the inactive
    ancestors of n(up to the
//first active ancestor y of n).
//**endof original optimizations**

//**My optimizations**
//Identify a Node by its Marking, and check existence in Act(fast) before iterating
    through all active nodes, when checking for coverage.

//10: Act:= Act \ { x | y     Ancestor_C(x) s.t. Lambda(y)  Lambda(n)    (y     Act     y
    \not\in Ancestor_C(n))};
//This is functionally equivalent to:
//"Remove from Act, all elements in { x  Act | Lambda(x)  Lambda(n)
//     ( y      inactiveAncestors(x) s.t. Lambda(y)  Lambda(n)   x  not  Ancestor(n)) }
// and their descendants."
//i.e. look only for the top nodes.

//Of all nodes to remove, check which one is the highest ancestor of the active node,
```

```
//determine the path to the active node, and make those(and their own inactive sets) the
    inactive set of the active node.
//Then remove all subtrees starting at such nodes.

//if n is deactivated, it is not in Act, popping it from Wait does nothing because of line
    5.
//remove it from Wait, when deactivating
#endregion

Act = new HashSet<Node>(Marking.EqualityComparer);
MTFHashSet wait = new MTFHashSet(initialMarking.Length);

//1: Let x0 be a new node such that \Lambda(x0) = m0;
Node node = new Node(initialMarking);
//2: X:= {x0}; Act:= X; Wait:= {(x0, t) | Lambda(x0) \fire{t} }; B:= ;
Act.Add(node);
wait.Add(node);

//3: while Wait \neq  do
while (wait.Count != 0)
{
    CancellationToken.ThrowIfCancellationRequested();

    //4: Pop( n , t) from Wait.
    node = (Node)wait.First();

    //5: if  n     Act then
    while (node.IsActive)
    {
        int t = node.next_tr++;
        if (t == transitions.Length)
        {
            wait.Remove(node);
            break;
        }
        if (!node.IsFireable(transitions[t]))
            continue;

        //6: m:= Post(Lambda(n), t);
        Marking original = node.Fire(transitions[t]);

        //7: Let n be a new node such that Lambda(n) = Acc(Lambda(Ancestor_C(n)  Act),m);
```

```
//8: X:= X    {n}; B:= B    {( n , t, n)};
Node newNode = new Node(original);
var list = node.ActiveAncestors;
foreach (var ancestor in list)
{
    if (ancestor < original)
        newNode.Pump(ancestor);
}

//9: if Lambda(n) is not covered by anything in Lambda(Act) then
if (!Act.Contains(newNode) && !Act.Any(act => act >= newNode))
{
    newNode.parent = node;

    //10: Act:= Act \ { x | y     Ancestor_C(x) s.t. Lambda(y)  Lambda(n)    (y
        Act     y \not\in Ancestor_C(n))};
    //This is functionally equivalent to:
    //"Remove from Act, all elements in { x  Act | Lambda(x)  Lambda(n)
    //     ( y      inactiveAncestors(x) s.t. Lambda(y)  Lambda(n)   x  not
        Ancestor(n)) }
    // and their descendants."
    Node[] removeThese = Act.Where(act => act <= newNode
        || (act.inactiveAncestors?.Any(anc => anc <= newNode) == true
            && !list.Contains(act)))
        .ToArray();

    //find highest removed ancestor
    int index_top = list.Count - 1;
    while (index_top >= 0 && !removeThese.Contains(list[index_top]))
        index_top--;

    //associate inactivated nodes with newNode
    if (index_top != -1)
    {
        var top = list[index_top];
        newNode.parent = top.parent;
        top.parent?.children.Add(newNode);
        top.parent?.children.Remove(top);

        List<Node> path = list.Take(index_top + 1).ToList();
```

```
                path.AddRange(path.Where(n => n.inactiveAncestors != null).SelectMany(n =>
                    n.inactiveAncestors).ToArray());
                newNode.inactiveAncestors = Minimize(path, newNode);
            }
            else
            {
                if (node.children == null)
                    node.children = new List<Node>();
                node.children.Add(newNode);
            }

            //remove the nodes and their descendants
            Stack<Node> stack = new Stack<Node>(removeThese);
            while (stack.Count != 0)
            {
                Node temp = stack.Pop();
                if (temp.IsActive)
                {
                    temp.IsActive = false;
                    Act.Remove(temp);
                    wait.Remove(temp);
                }
                if (temp.children != null)
                {
                    foreach (Node child in temp.children)
                        stack.Push(child);
                    temp.children.Clear();
                }
                temp.inactiveAncestors?.Clear();
                temp.parent = null;

            }

            //11: Act:= Act   {n}; Wait:= Wait   {(n, u) | \Lambda(n) \fire{u} };
            Act.Add(newNode);
            newNode.IsActive = true;
            wait.Add(newNode);
            CurrentMemory = Act.Count; //not counting associated inactive ancestors, there
                can't be many
            break;
        }//endif newNode not covered by Act
    }//while node in Act
```

```
            if (!node.IsActive)
                wait.Remove(node);
        }//while Wait.Count != 0
        //15: Return C = (X, x0, B, \Lambda) and Act.

        return;

    }

    class Node : Marking
    {
        public Node parent;
        public List<Node> children;
        public List<Node> inactiveAncestors;
        public bool IsActive = true; //A hashset is fast, a field is even faster
        public int next_tr;
        List<Node> m_ancestors;
        List<Node> m_activeAncestors;

        public Node(Marking label) : base(label) { }

        public new Node Fire(Transition t)
        {
            return new Node(supply.Add(t.effect));
        }
        /// <summary>
        /// Enumerates from self to root.
        /// </summary>
        /// <returns></returns>
        public List<Node> Ancestors
        {
            get
            {
                if (m_ancestors == null)
                {
                    m_ancestors = new List<Node>();
                    Node walker = this;
                    while (walker != null)
                    {
                        m_ancestors.Add(walker);
                        if (inactiveAncestors != null)
                            foreach (var inactive in inactiveAncestors)
```

```
                        m_ancestors.Add(inactive);
                    walker = walker.parent;
                }
            }
            return m_ancestors;
        }
    }
    /// <summary>
    /// Enumerates from self to root.
    /// </summary>
    /// <returns></returns>
    public List<Node> ActiveAncestors
    {
        get
        {
            if (m_activeAncestors == null)
            {
                m_activeAncestors = new List<Node>();
                Node walker = this;
                while (walker != null)
                {
                    m_activeAncestors.Add(walker);
                    walker = walker.parent;
                }
            }
            return m_activeAncestors;
        }
    }

}//Node

List<Node> Minimize(List<Node> markings, Node anchor)
{
    var temp = markings.ToList();
    for (int i = temp.Count - 1; i >= 0; i--)
    {
        if (temp[i] < anchor
         || temp.Any(m => !ReferenceEquals(m, temp[i]) && m >= temp[i]))
            temp.Remove(temp[i]);
    }
    return temp;
}
```

```
    }
}
```

A.6 HanVal.cs

```
 using  System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace PetriTester
{
    //This algorithm is defined on a rather low level.
    //Making a _Simple version would at most change the way "W:= {MO} x T" and the back pointers work.
    //Thus, make a DFS and a MTF version instead, to mirror the results in the paper.
    //The parallel version will parallelize the MTF version.
    class HanVal_MTF: Grapher
    {
        class Node : Marking
        {
            public int next_tr = 0;

            public new Node Fire(Transition t)
            {
                return new Node(supply.Add(t.effect));
            }
            public Node(Marking marking) : base(marking) { }
            public Node(int?[] supply) : base(supply) { }
        }

        HashSet<Node> active;
        Dictionary<Node, List<Node>> back;

        override protected IEnumerable<Marking> GetFinalMarkings()
        {
            return active;
        }

        override public void Run(CancellationToken CancellationToken)
```

```
{
    #region Original Description
    /*
     *  1 F := {M0}; A := {M0}; W := {M0}  T ; M0.B := nil
        2 while W not= \emptyset do
        3 (M,t) := any element of W; W := W \ {(M,t)}
        4 if  M .isfireable(t) then continue
        5 M' := the  -marking such that M\fire(t)M'
        6 if M'    F then continue
        7 Add- (M,M')
        8 if    was added then if M'   F then continue
        9 Cover-check(M') // may update A and W
        10 if M' is covered then continue
        11 F := F    {M'}; A := A    {M'}; W := W    ({M'}   T ); M'.B := M
     *
     *  Add- (M,M') //basic
        1 last := M; now := M; added := False
        2 repeat
        3     if now < M'     p      P : now(p) < M'(p) <  then
        4         added := True; last := now
        5         for each p    P such that now(p) < M'(p) <  do
        6             M'(p) :=
        7     if now.B = nil then now := M else now := now.B
        8 until now = last
     *
     * This is only the basic addOmega, add history merged scanning to it in our
         implementation.
     * Paper says something about a global search_now that is assigned to node.search_nr to
         identify the last node used for pumping.
     * I have no idea why you wouldn't just compare a node's pointer to the stored value of
         this pointer.
     * Let's just do that. Or in C#, Node last = now. Exactly like in the code.
     */
    #endregion
    /*
    * Merge histories if an identical marking is found, or covered by a marking that is
        identical except for \omega tokens.
    *
    * PROOF OF CORRECTNESS
    * Let M' > M'' in this specific way, having an \omega token where M'' does not in places P'
    * We would merge the history of M'' with M' to facilitate faster pumping. So let such a
        pumping take place.
```

```
* Let N be an ancestor of M'', N [\sigma> M'', and M' [\tau> N' > N.
*
* Were we not to have merged, we would continue.
* Let D s.t. D(p) = 0 if N(p) = N'(p), D(p) = -\omega if N'(p) = \omega != N(p), and D(p)
    = 1 otherwise.
* The effect of \sigma\tau is at least D.
* The effect of \tau\sigma is at least D.
*
* Note that D(p) = 1 in at least places P'.
* If M'' does not have an \omega token, neither does N.
* If M' has an \omega token, so does N'.
*
* N' >= N, \sigma is fireable in N, so it is fireable in N'. Let N'[\sigma> M'''
* M'[\tau\sigma> M'''. M''' >= M'+ D. So pumping takes place in places P'.
* END OF PROOF
*
*/
//in short: KnM, no pruning, merged history and repeated history scanning.
//but it remembers found markings so as to not redo an explored subtree.

back = new Dictionary<Node, List<Node>>(Marking.EqualityComparer);
active = new HashSet<Node>();
MTFHashSet workingSet = new MTFHashSet(initialMarking.Length);

//1 F := {M0}; A := {M0}; W := {M0}  T ; M0.B := nil
Node node = new Node(initialMarking);
Node newNode;
List<Node> list;

back.Add(node, new List<Node>());
active.Add(node);
workingSet.Add(node);
CurrentMemory = 1;

//2 while W not=   do
while (workingSet.Count != 0)
{
    //3 (M, t) := any element of W; W:= W \ { (M, t)}
    for (node = (Node)workingSet.Pop(); node.next_tr < transitions.Length; node.next_tr++)
    {
        CancellationToken.ThrowIfCancellationRequested();
        Transition t = transitions[node.next_tr];
```

```
//4 if  M .isfireable(t) then continue
if (!node.IsFireable(t))
    continue;

//5 M' := the  -marking such that M\fire(t)M'
newNode = node.Fire(t);

//6 if M'    F then continue
if (back.TryGetValue(newNode, out list))
{
    list.Add(node);
    continue;
}

//7 Add- (M, M')
if (AddOmega(node, newNode))
    //8 if    was added then if M'   F then continue
    if (back.TryGetValue(newNode, out list))
    {
        list.Add(node);
        continue;
    }

//9 Cover - check(M') // may update A and W
bool largerFound = false;
list = new List<Node>();
#region Cover-check(M')
bool lesserFound = false;
foreach (Node act in active.ToArray())
{
    if (!lesserFound)
    {
        int? result = Marking.Compare(newNode, act);
        if (result != null)
        {
            if (result < 0) //newMarking < act
            {
                largerFound = true;
                if (AlmostEqual(newNode, act))
                    back[act].Add(node);
                continue;
            }
```

```
                    if (result > 0) //newMarking > act
                    {
                        lesserFound = true;
                        if (AlmostEqual(act, newNode))
                            list.AddRange(back[act]);
                        active.Remove(act);
                        act.next_tr = transitions.Length;
                    }
                }
            }
            else
            {
                //to check if act is strictly covered by marking is a smaller check than
                    comparing them completely
                if (act < newNode)
                {
                    if (AlmostEqual(act, newNode))
                        list.AddRange(back[act]);
                    active.Remove(act);
                    act.next_tr = transitions.Length;
                }
            }
        }//foreach act
        #endregion
        //10 if M' is covered then continue
        if (largerFound)
            continue;

        //11 F:= F    { M'}; A := A    {M'}; W:= W    ({M'}    T ); M'.B := M
        list.Add(node);
        back.Add(newNode, list);
        active.Add(newNode);
        workingSet.Add(newNode);
        CurrentMemory++;
    }//for transitions
}//while WorkingSet not empty
}//Run

bool AddOmega(Node root, Node newMarking)
{
    //*add -  (m, m') //basic
    //1 last:= m; now:= m; added:= false
```

```
//2 repeat
//3    if now < m'     p      p : now(p) < m'(p) <  then
//4        added:= true; last:= now
//5        for each p    p such that now(p) < m'(p) <  do
//6            m'(p) :=
//7    if now.b = nil then now := m else now:= now.b
//8 until now = last
//adapt this to back being a list of markings

//create ancestorset, removes duplicates
//this may seem overkill, but the ancestorset can be as big as the found set, so making a
    distinct List in O(n^2) can be bad
HashSet<Node> ancestors = new HashSet<Node>();
Queue<Node> todo = new Queue<Node>();
Node current;
todo.Enqueue(root);
while (todo.Count != 0)
{
    current = todo.Dequeue();
    if (ancestors.Add(current))
        foreach (Node parent in back[current])
            todo.Enqueue(parent);
}

//loop through all ancestors until no change happened for an entire loop
Node last = ancestors.First();
bool added = false;
bool firstrun = true;
while (true)
{
    foreach (Node now in ancestors)
    {
        if (!firstrun && now == last)
            return added;
        if (newMarking > now && newMarking.Pump(now))
        {
            added = true;
            last = now;
        }
    }
    firstrun = false;
}
```

```
        }//AddOmega

        bool AlmostEqual(Node smaller, Node greater)
        {
            for (int i = 0; i < smaller.Length; i++)
                if (greater[i] == null || greater[i] == smaller[i])
                    return false;
            return true;
        }
    }
}
```

Appendix B

Benchmark images

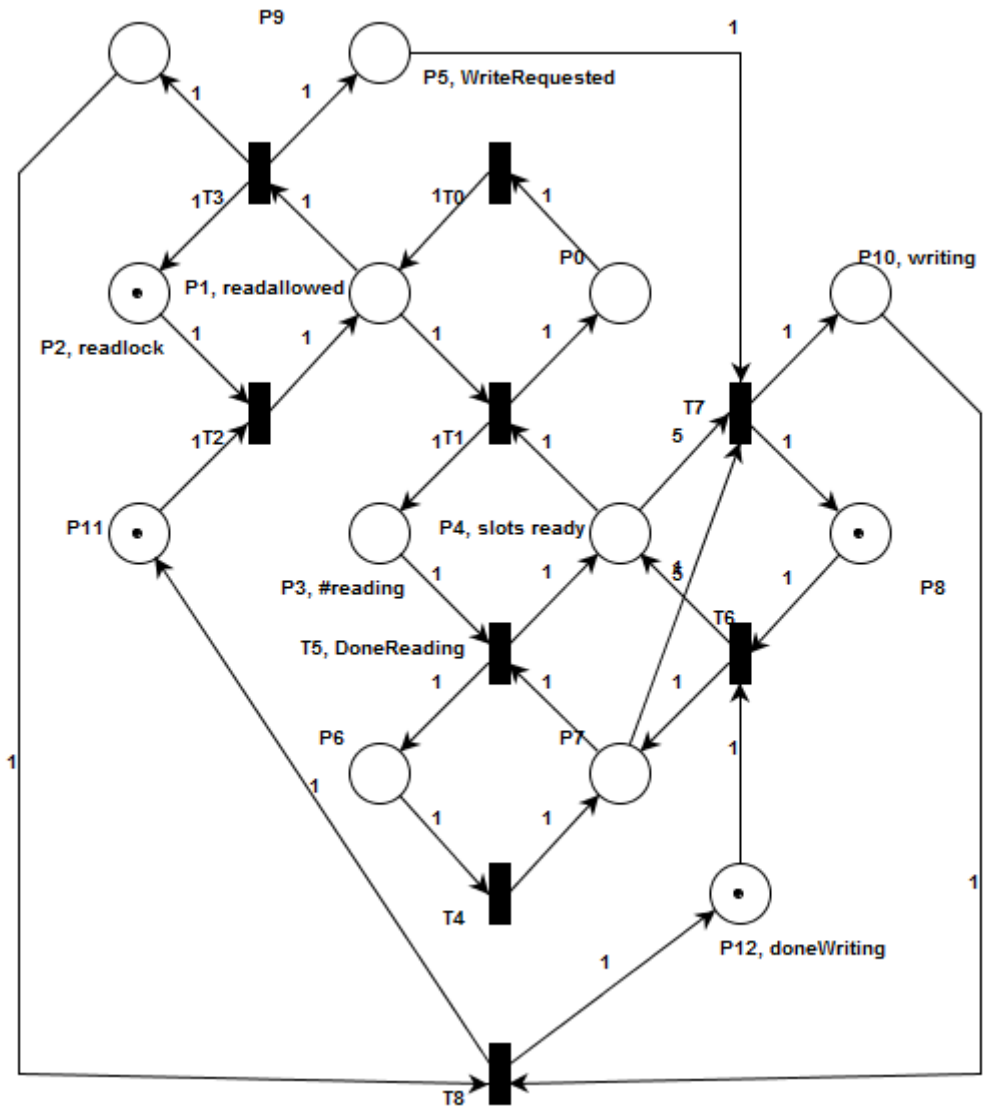


Figure B.1: The Petri net read-write.

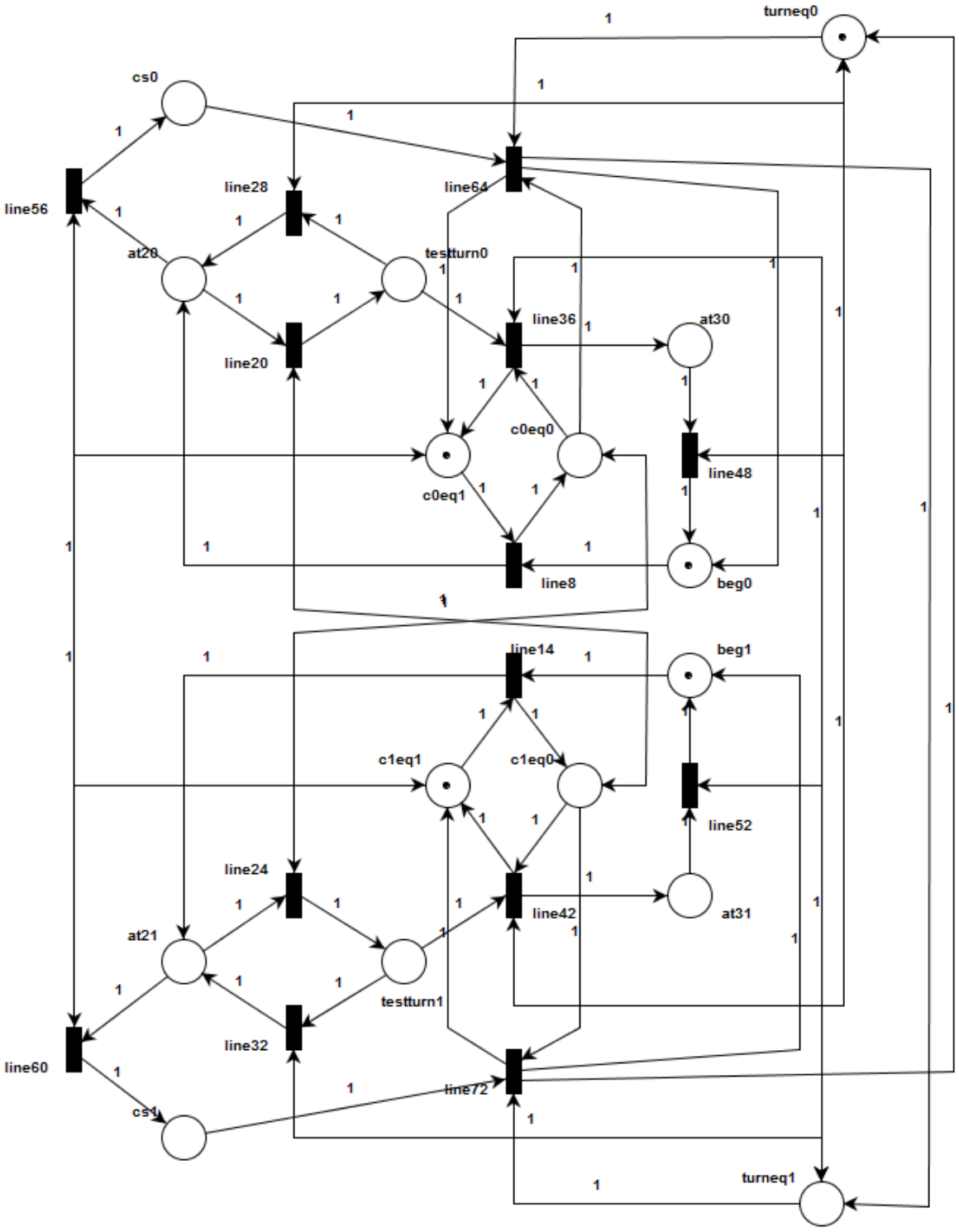


Figure B.2: The Petri net newdekker.

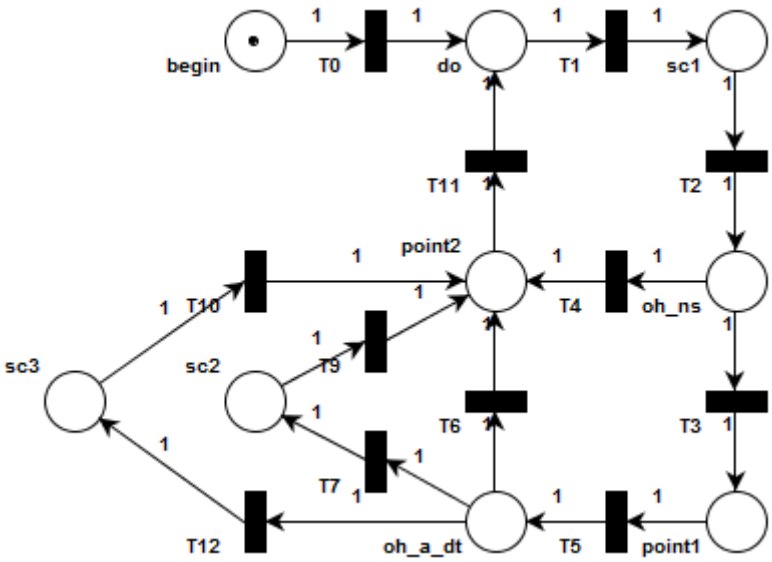


Figure B.3: The Petri net newrtp.

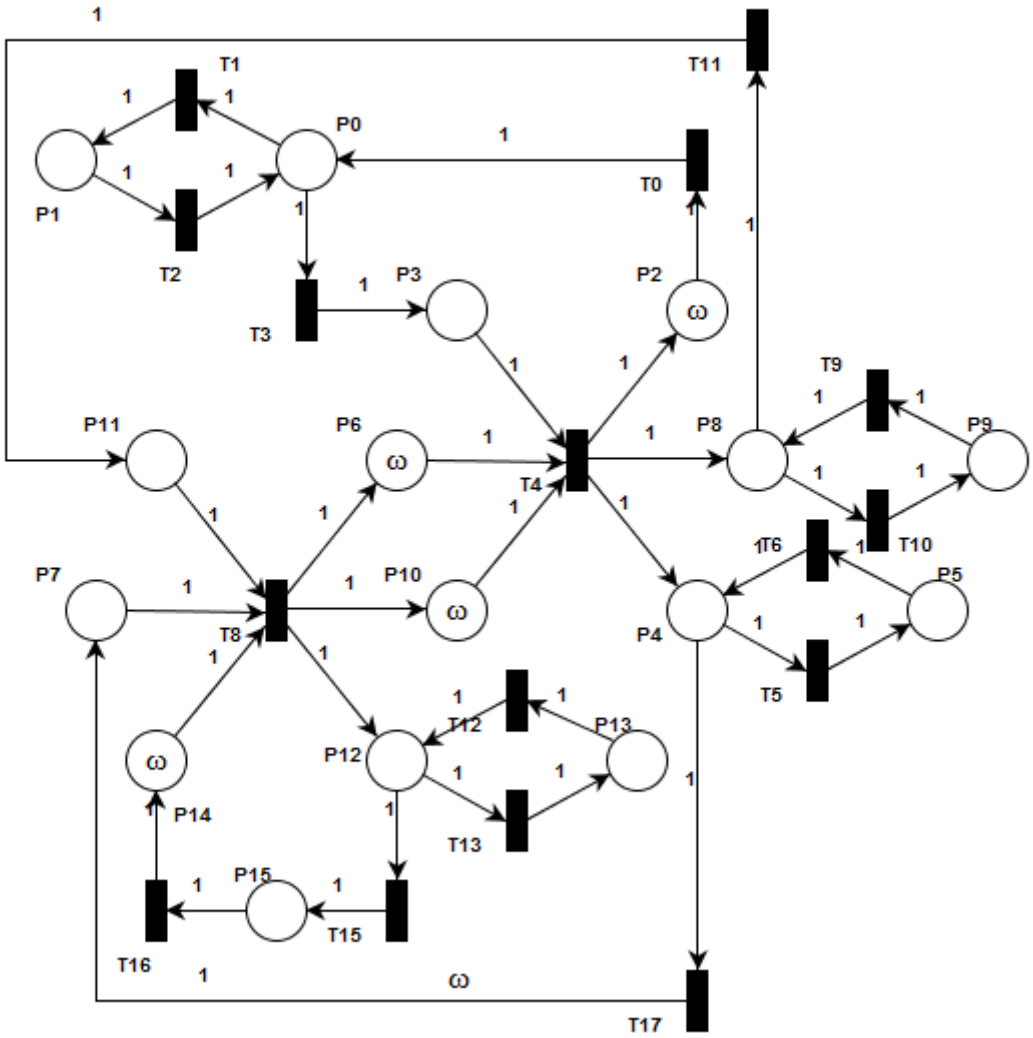


Figure B.4: The Petri net kanban.

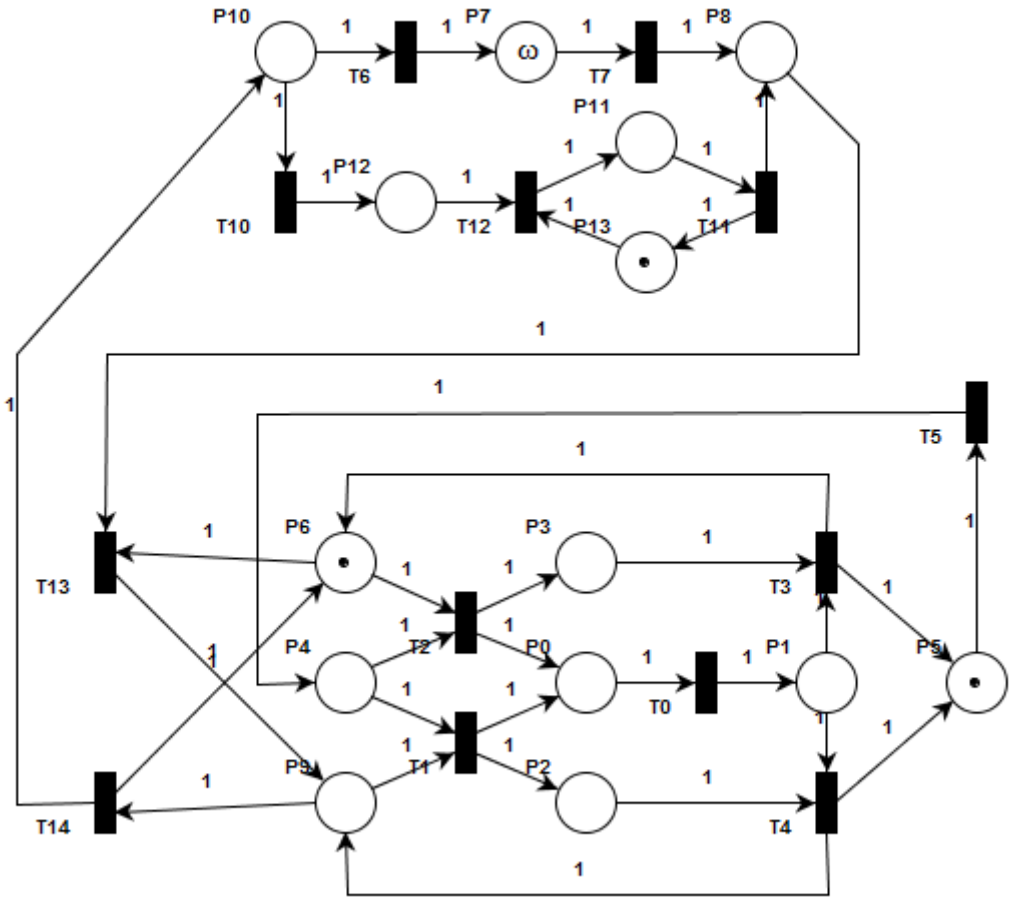


Figure B.5: The Petri net csm.

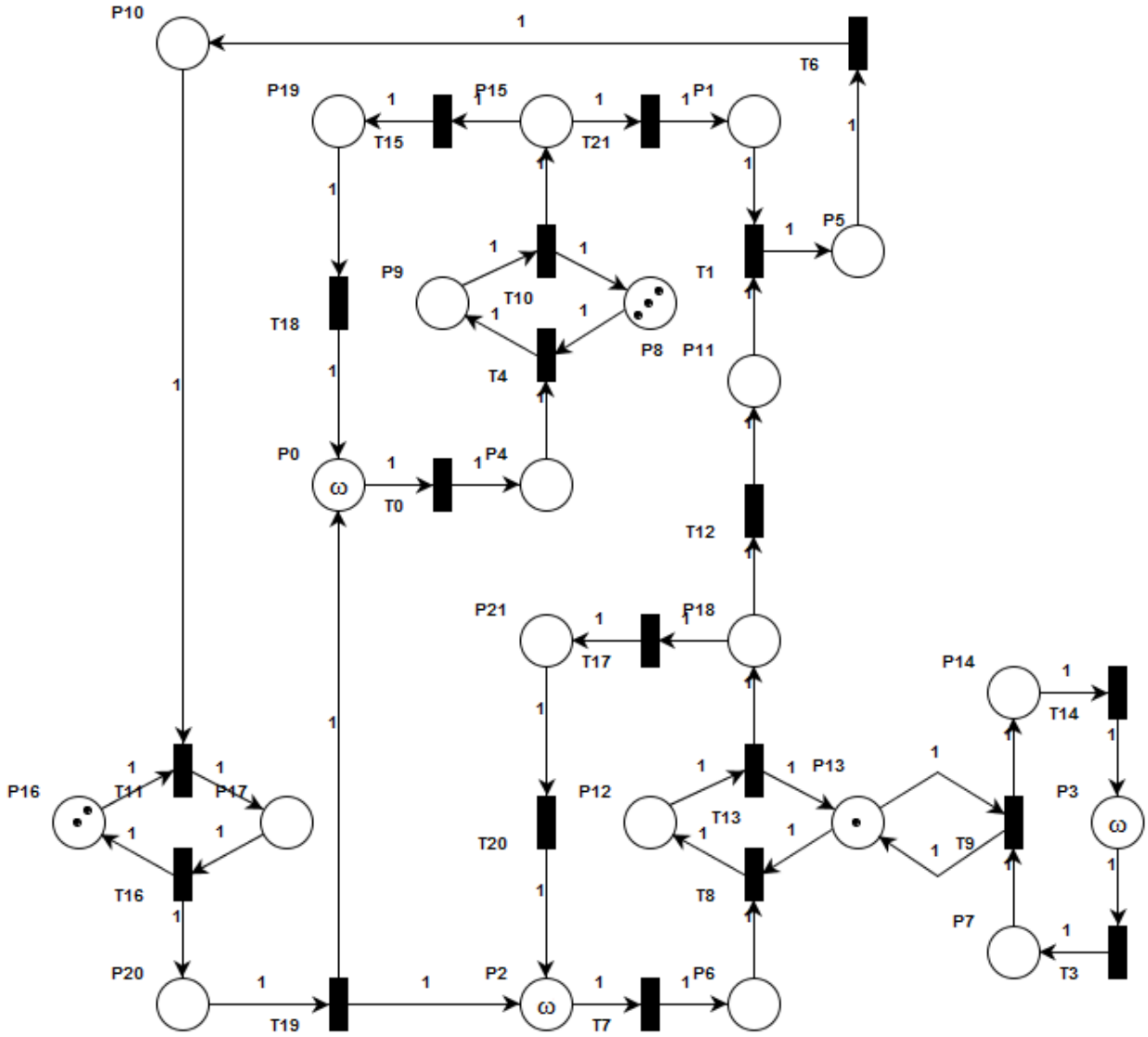


Figure B.6: The Petri net fms.

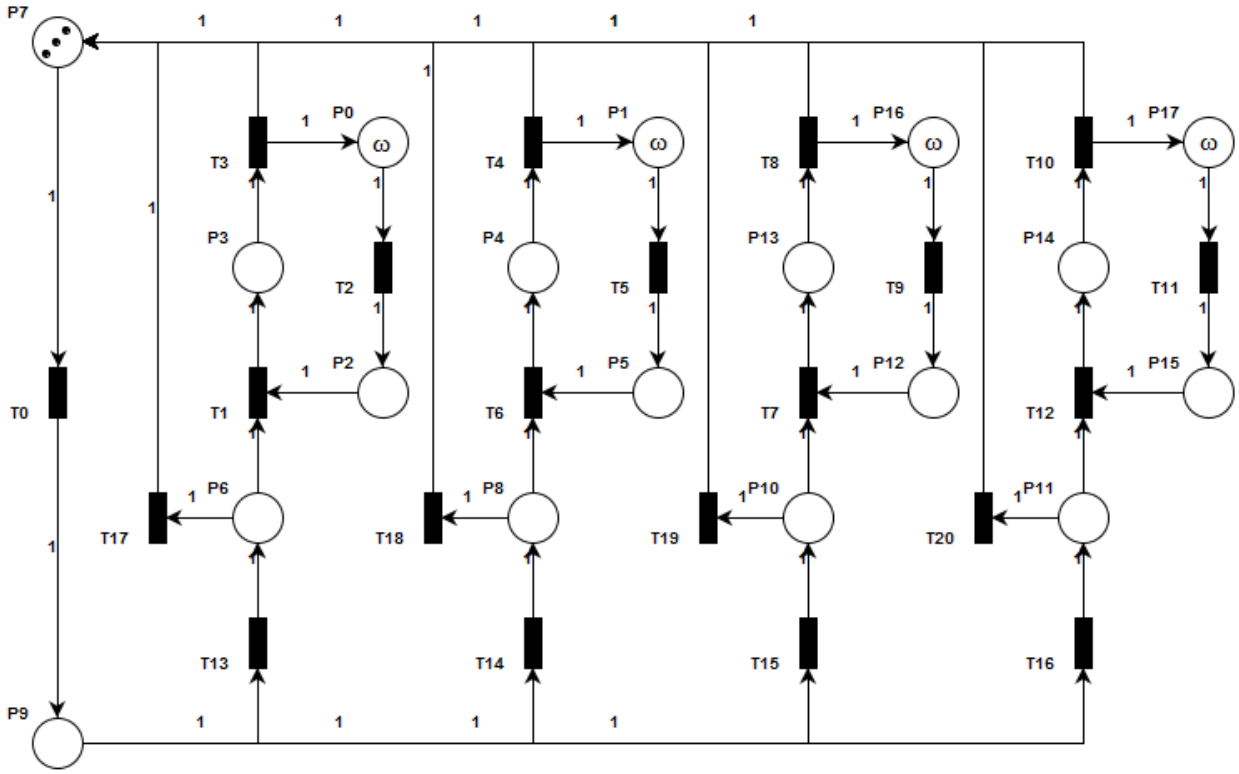


Figure B.7: The Petri net multipool.

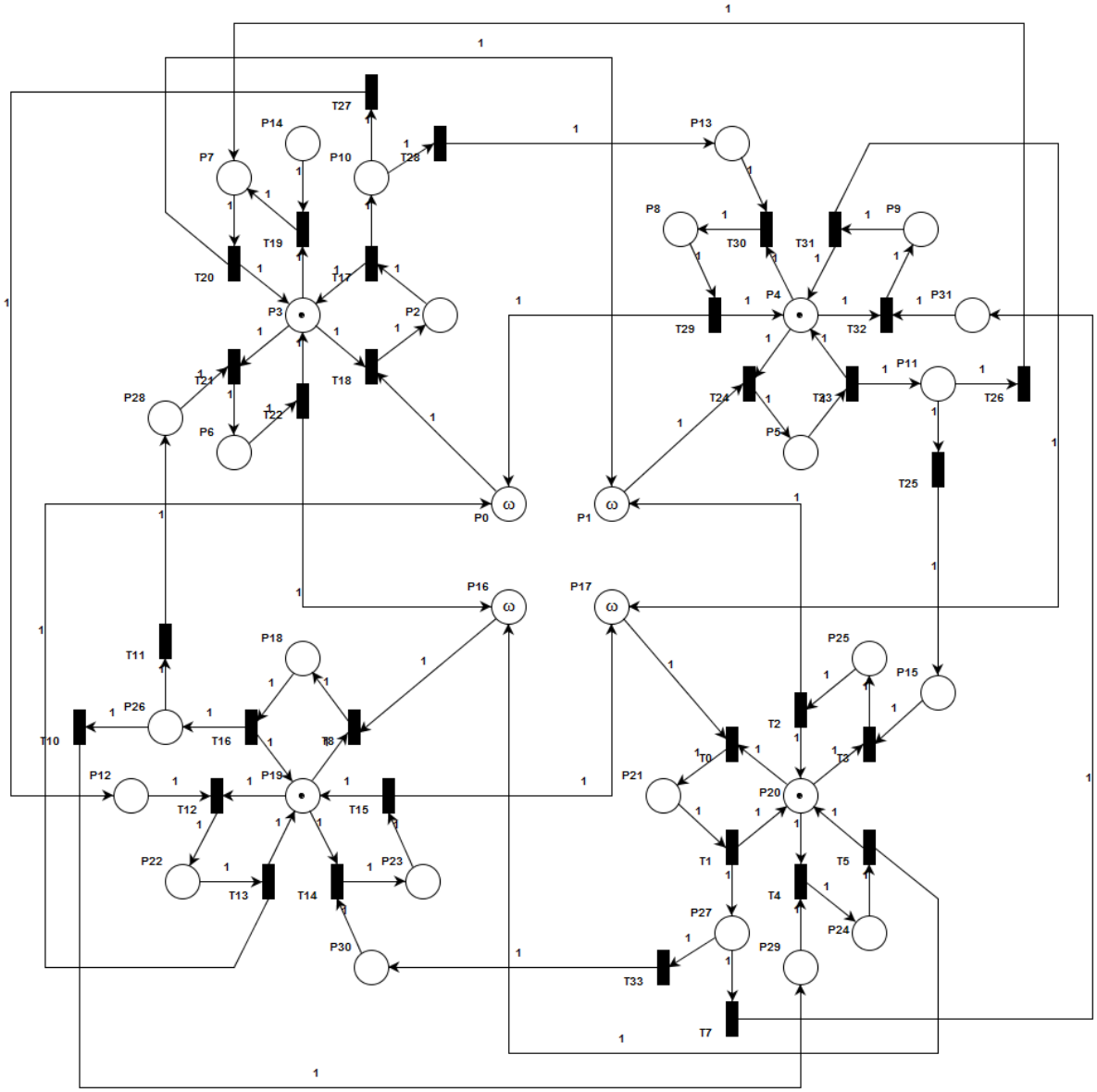


Figure B.8: The Petri net mesh2x2.

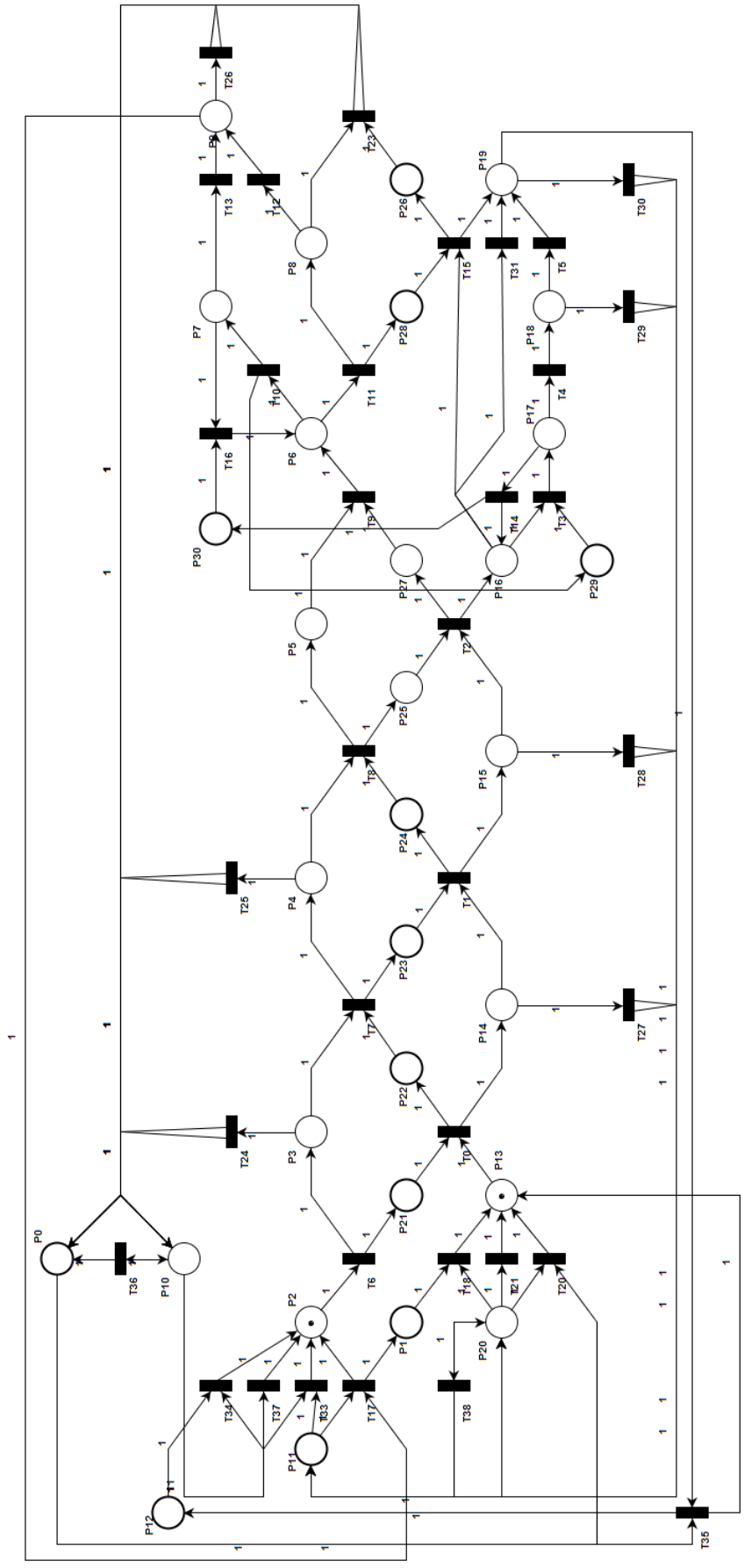


Figure B.9: The Petri net pncsacover. Bolded places can get an ω token during execution.

Bibliography

[AK76] Toshiro Araki and Tadao Kasami. Some decision problems related to the reachability problem for petri nets. *Theoretical Computer Science*, 3(1):85 – 104, 1976.

[Bak73] Henry Baker. Rabin's proof of the undecidability of the reachability set inclusion problem of vector addition systems. 1973.

[BR09] Wilfried Brauer and Wolfgang Reisig. Carl adam petri and "petri nets". *Fundamental Concepts in Computer Science*, 3:129–139, 2009.

[Dic13] Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4):413–422, 1913.

[Fin91] Alain Finkel. The minimal coverability graph for petri nets. In *Advances in Petri Nets 1993, Papers from the 12th International Conference on Applications and Theory of Petri Nets, Gjern, Denmark, June 1991*, pages 210–243, 1991.

[GRB10] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set of petri nets. *Int. J. Found. Comput. Sci.*, 21(2):135–165, 2010.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.

[Reu90] Christophe Reutenauer. *The Mathematics of Petri Nets*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[RS13] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for petri nets: Karp and miller algorithm with pruning. *Fundam. Inform.*, 122(1-2):1–30, 2013.

[VH12] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. In *Application and Theory of Petri Nets - 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25-29, 2012. Proceedings*, pages 208–227, 2012.

[VH14] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. *Fundam. Inform.*, 131(1):1–25, 2014.

[VVN81] Rüdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3):299 – 325, 1981.