SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Backwards Reachability in Petri Nets Using Weakly-Acyclic DFAs

Xinyi Cui

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Backwards Reachability in Petri Nets Using Weakly-Acyclic DFAs

Backwards Reachability in Petrinetzen mit fast azyklischen DFAs

Author:	Xinyi Cui
Supervisor:	Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	M. Sc. Philipp Czerner
Submission Date:	15.03.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.03.2024 Xinyi Cui

Acknowledgments

I am very grateful to Prof. Javier Esparza and Prof. Michael Blondin for giving me the opportunity to work on such a new and interesting topic and for their valuable advice. I want to deeply thank my advisor Philipp Czerner for his wonderful guidance and constructive feedback throughout the whole process. Lastly, I want to thank my family and friends for all their support and encouragement.

Abstract

The coverability problem of Petri nets plays a crucial role in analyzing and arguing about distributed and concurrent systems over various application areas, including parallel programs, biological models, and business processes. Nevertheless, its EXPSPACE-complete nature presents a challenge when confronted with practical scenarios.

This thesis presents a novel approach to the coverability problem, which utilizes a class of formal language called weakly acyclic. We design a data structure for minimal weakly acyclic DFAs, this can be seen as a generalization of binary decision diagrams. Subsequently, we integrate this data structure into the Backwards Reachability Algorithm, an existing high-level procedure for solving the coverability problem.

We perform benchmarks on real-life instances, which show that our implementation is competitive with prevailing coverability tools and that the representation with weakly acyclic languages provides a compact finite description of configurations in Petri nets.

Contents

1 Introduction

Petri nets are a useful mathematical model for describing and analyzing concurrent systems. Their generality makes them applicable in a wide range of areas, including the verification of concurrent programs [13, 15], manufacturing and control systems [18, 19], biological systems [4, 5], and business processes [14].

A Petri net consists of places that are connected by transitions. The places contain tokens, which can move between the places by executing transitions. We call a marking a configuration of the Petri net, where each place is assigned a specific amount of tokens.

This formal model allows us to examine interesting properties on an abstract mathematical level. Some of the most studied decision problems in this area are about the reachability and boundedness of Petri nets. The reachability problem investigates which markings are reachable from a given start marking by executing transitions of the net. The boundedness problem studies, if the number of tokens in the places is limited by an upper bound for all reachable markings.

In this thesis, we shift the attention to another important decision problem, the coverability problem of Petri nets, which investigates the following issue: Can a given start marking reach a marking larger than a given target marking by executing an arbitrary amount of transitions of the Petri net? Here, a larger marking means that this marking assigns at least as many tokens to the place as the target marking.

There exist various decision procedures for the coverability problem. For instance, the Coverability Graph Algorithm builds the coverability graph of a given Petri net and start marking, which can be efficiently checked for coverability given any target marking [10]. Rackoff's Algorithm is another procedure, which makes use of the reachability graph by constructing it until a specific depth dependent on the target marking [17, 9].

The decision procedure we are examining for the thesis is the Backwards Reachability Algorithm. This algorithm starts with the infinite set of all markings larger than the target marking. Such a set with the property that if a certain element is included, all elements greater than it are also included is called upward closed. By iteratively computing and assembling all possible predecessor markings of this upward closed set and checking if the predecessors contain the start marking, we obtain the solution to the coverability problem [1].

To represent the infinite upward closed sets of the Backwards Reachability Algorithm, common approaches operate with the unique finite set of minimal markings, which every upward closed set possesses. However, the efficiency of this algorithm depends heavily on the number of minimal markings for the upward closed sets, which might grow exponentially with the number of places in the Petri net [7]. To address this issue, studies have been conducted on data structures designed for representing infinite sets [20, 16, 8].

In this thesis, we want to investigate a novel approach to the Backwards Reachability Algorithm by representing the infinite upward closed sets with formal languages called weakly acyclic. This class of weakly acyclic languages is recognized by deterministic finite automata which contain no simple cycle except for self-loops. By mapping a marking to a word over an alphabet and a set of markings to a language over that alphabet, we are able to describe the infinite upward closed sets from the algorithm with the use of these weakly acyclic automata.

In detail, we make the following contributions:

- We introduce the class of weakly acyclic languages, which are a subset of regular languages (Chapter 3),
- we design a data structure called Table of Nodes for representing weakly acyclic automata efficiently (Chapter 4),
- we implement important operations on this data structure (Chapter 4),
- we define a mapping between weakly acyclic automata and markings of Petri nets (Chapter 5),
- we construct a transducer – an automaton with input and output – from a Petri net (Chapter 5),
- we integrate the Table of Nodes into the Backwards Reachability Algorithm (Chapter 5), and
- we perform benchmarks on our implementation and compare it to other coverability tools (Chapter 6).

The benchmarks show that our approach can characterize the infinite sets of markings compactly with the data structure we developed. Furthermore, comparisons with existing tools emphasize that our implementation is able to compete with state-of-the-art solvers for the coverability problem.

2 Preliminaries

This chapter aims to define a standardized formal notation of the theoretical concepts used throughout the thesis.

2.1 Formal Languages

DFA

A *deterministic finite automaton* (DFA) is a tuple $A = (Q, \Sigma, q_0, \delta, F)$ where

- Q is a finite set of states;
- Σ is a finite alphabet;
- $q_0 \in Q$ is an initial state;
- $\delta : Q \times \Sigma \to Q$ is a transition function;
- $F \subseteq Q$ is a set of accepting final states.

NFA

A *nondeterministic finite automaton* (NFA) is a tuple $B = (Q, \Sigma, q_0, \delta, F)$ where all elements are equivalently defined as for a DFA except the transition function:

- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ where $\mathcal{P}(Q)$ denotes the power set of all states Q.

Residual Language

The *residual language* of the language L with respect to a letter a is defined as $L^a = \{w \in \Sigma^* : aw \in L\}$.

We can extend this definition to the residual language with respect to a word $v \in \Sigma^*$ to $L^v = \{w \in \Sigma^* : vw \in L\}$.

Language of an Automaton State

For a state q in an automaton A, we define the language $L_A(q)$ containing all words accepted by A with initial state q.

Transducer

A *transducer* $\mathcal{T}$ over Σ is an NFA over the alphabet $\Sigma \times \Sigma$. It is a tuple $\mathcal{T} = (Q, \Sigma, q_0, \delta, F)$ where

- Q is a finite set of states;
- Σ is a finite alphabet;
- $q_0 \in Q$ is an initial state;
- $\delta : Q \times (\Sigma \times \Sigma) \to \mathcal{P}(Q)$ is a transition function;
- $F \subseteq Q$ is a set of accepting final states.

$\mathcal{T}$ accepts the pair of words (w, v) with $w = w_1 w_2 \ldots w_n$ and $v = v_1 v_2 \ldots v_n$, if it accepts $(w_1, v_1)(w_2, v_2) \ldots (w_n, v_n)$. We call the set of all w the preimage and the set of all v the image of transducer $\mathcal{T}$.

2.2 Petri Nets

A *Petri net* N is a tuple (P, Tr, F) where

- P is a finite set of *places*;
- Tr is a finite set of *transitions* disjoint from S;
- $F \subseteq (P \times Tr \times \mathbb{N}) \cup (Tr \times P \times \mathbb{N})$ is a *flow relation*.

In drawings of Petri nets, the places are usually represented by circles, the transitions by rectangles, and the flow relation is shown by numbered arrows between places and transitions. Figure 2.1 shows an example with three places (p_1, p_2, p_3) and two transitions (t_1, t_2). For this instance, the flow relation contains the following elements: $(p_1, t_1, 2), (t_1, p_2, 1), (t_1, p_3, 2)$, $(p_2, t_2, 2), (p_3, t_2, 1)$, and $(t_2, p_1, 3)$.

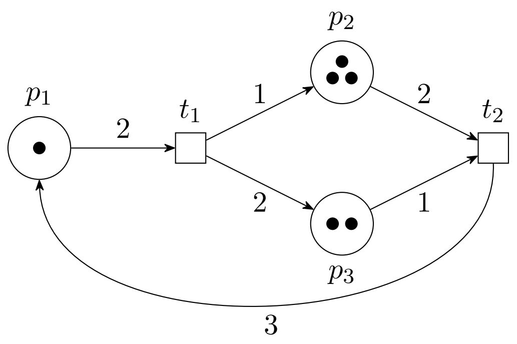


Figure 2.1: Example of a Petri Net

Marking

A *marking* of a Petri net is a mapping $M : P \to \mathbb{N}$ from the places of the net to the natural numbers, assigning each place a certain amount of tokens. In drawings, the markings are commonly represented by black dots inside the places, like in Figure 2.1.

Firing Rule

A transition $t \in \mathit{Tr}$ is *enabled* at a marking M if for all tuples $(p, t, n) \in F$, marking M assigns at least n tokens to place p.

If a transition t is enabled at marking M, it can *fire* leading from M to another marking M', which we also denote by $M \xrightarrow{t} M'$. The resulting M' is of the following form:

$$M'(p) = \begin{cases} M(p) + (m - n) & \text{if } \exists(p,t,n) \in F \text{ and } \exists(t,p,m) \in F \\ M(p) - n & \text{if } \exists(p,t,n) \in F \text{ and } \nexists(t,p,m) \in F \\ M(p) + m & \text{if } \nexists(p,t,n) \in F \text{ and } \exists(t,p,m) \in F \\ M(p) & \text{otherwise} \end{cases}$$

Reachable Markings

A sequence of transitions $\sigma = t_1 \dots t_n$ is enabled at a marking M, if there are markings $M_1, M_2, \dots, M_n$ such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \dots \xrightarrow{t_n} M_n$. We denote this by $M \xrightarrow{\sigma} M_n$.

If $M \xrightarrow{\sigma} M'$ for some markings M, M' and some sequence σ, we say that M' is *reachable* from M and denote this by $M \xrightarrow{*} M'$.

Upward Closed Set of Markings

For markings M and M', we write $M \geq M'$ if for every place p of a Petri net $M(p) \geq M'(p)$. This means that M assigns at least as many tokens to every state as M'. We say that marking M *covers* marking M'.

A set $\mathcal{M}$ of markings of the Petri net N is *upward closed* if $M \in \mathcal{M}$ and $M' \geq M$ imply $M' \in \mathcal{M}$. In other words, if a marking exists in the upward closed set, then all other markings which cover this marking are also part of the set.

A marking M of an upward closed set $\mathcal{M}$ is minimal if $\nexists M' \in \mathcal{M}$ such that $M' \leq M$. Every upward closed set $\mathcal{M}$ can be uniquely identified by a finite set of minimal elements.

3 Weakly Acyclic Languages

In this chapter, we introduce a class of formal languages called weakly acyclic. They are a subset of the regular languages and are characterized by having no cycles beyond self-loops in the automaton representation.

3.1 Weakly Acyclic DFAs

Definition 1 ([3]). *Let $A = (Q, \Sigma, q_0, \delta, F)$ be a DFA and let $\alpha(w)$ be the set of letters which occur within in word w. A is weakly acyclic if $\delta(q, w) = q$ implies $\delta(q, a) = q$ for every $a \in \alpha(w)$*

The following definitions are equivalently expressing that a DFA $(Q, \Sigma, q_0, \delta, F)$ is *weakly acyclic*:

– The binary relation $\precsim$ over $Q \times Q$ with $q \precsim q'$ if $\delta(q, w) = q'$ is a partial order.
– Each strongly connected component of underlying directed graph contains a single state.
– The underlying directed graph does not contain any simple cycle except from self-loops.

Figure 3.1 shows an example for a weakly acyclic DFA and Figure 3.3 illustrates two not weakly acyclic DFAs.

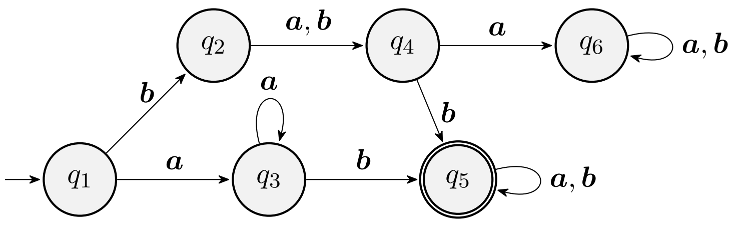


Figure 3.1: Example of a Weakly Acyclic DFA

Lemma 1. *Let A be a weakly acyclic DFA. The minimal DFA that accepts $L(A)$ is also weakly acyclic.*

Proof. See [3, Proposition 4]. $\square$

3.2 Other Representations

As with regular languages, there are equivalent ways to represent weakly acyclic languages. Let $\alpha(w)$ be the set of letters which occur within in word w. Blondin *et al.* have shown that weakly acyclic DFAs, weakly acyclic NFAs, and weakly acyclic expressions represent the same class of weakly acyclic languages [3].

3.2.1 Weakly Acyclic NFAs

An NFA $(Q, \Sigma, q_0, \delta, F)$ is *weakly acyclic* if

- $q \in \delta(q, w)$ implies $\delta(q, a) = q$ for every $a \in \alpha(w)$;
- the underlying directed graph does not contain any simple cycle except from self-loops and nondeterminism with a letter a can only appear from a state with no self-loop of a.

3.2.2 Weakly Acyclic Expressions

The class of weakly acyclic languages can also be characterized by *weakly acyclic expressions* of the following form:

$$r ::= \emptyset \mid \Gamma^* \mid \Lambda^* a r \mid r + r \quad \text{where} \quad \Gamma, \Lambda \subseteq \Sigma \quad \text{and} \quad a \in \Sigma \setminus \Lambda$$

3.3 Properties

This section describes some relevant properties of weakly acyclic languages, which have been shown by Blondin *et al.* [3].

3.3.1 Position in Language Hierarchy

Weakly acyclic languages lie strictly in between finite and regular languages, as illustrated in Figure 3.2.

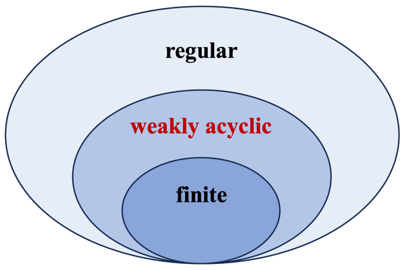


Figure 3.2: Weakly Acyclic Class in Language Hierarchy

They are within the regular languages, as they can be characterized with DFAs, NFAs, and regular expressions. Furthermore, every finite language is weakly acyclic, as only a self-loop in the trap state is necessary to represent a finite set of words.

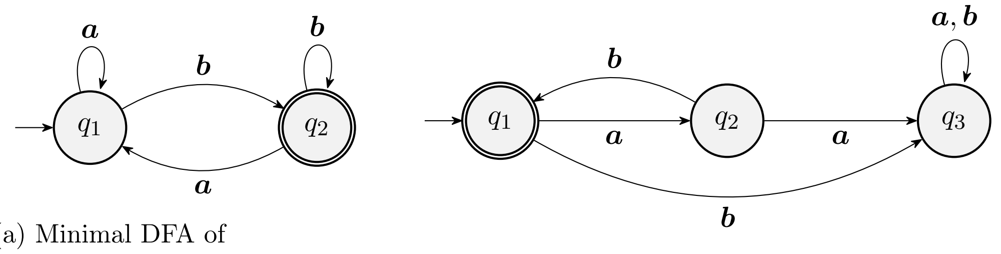
3.3.2 Closure Properties

Weakly acyclic languages are closed under union, intersection, and complementation but not under concatenation or Kleene star.

As weakly acyclic languages can be described by DFAs, complementing a DFA still preserves its weakly acyclic structure. Furthermore, union is already present in the definition of weakly acyclic expressions in Subsection 3.2.2. Therefore, this class is also closed under intersection, which can be expressed with the combination of union and complementation.

Weakly acyclic languages are not closed under concatenation. The expressions $(\boldsymbol{a}+\boldsymbol{b})^*$ and $\boldsymbol{b}$ are weakly acyclic by themselves. The language of their concatenation $(\boldsymbol{a}+\boldsymbol{b})^*\boldsymbol{b}$ is depicted in its unique minimal DFA in Figure 3.3a.

By Lemma 1 the minimal automaton of weakly acyclic language remains weakly acyclic. As this automaton contains a cycle of length 2, this language is not weakly acyclic. The same argumentation can be made for the Kleene star closure, with $\boldsymbol{ab}$ being a weakly acyclic language but $(\boldsymbol{ab})^*$, represented in Figure 3.3b, is not weakly acyclic.



(a) Minimal DFA of $L((\boldsymbol{a}+\boldsymbol{b})^*\boldsymbol{b})$

(b) Minimal DFA of $L((\boldsymbol{ab})^*)$

Figure 3.3: Not Weakly Acyclic DFAs

4 A Data Structure for Weakly Acyclic DFAs

In order to use weakly acyclic languages efficiently, a data structure for storing the automata and offering language operations is required. This chapter introduces a possible implementation approach for the data structure and presents algorithms for interesting operations.

4.1 Master Automaton

We first take a look at how multiple weakly acyclic DFAs can be combined in theory by establishing the concept of the master automaton.

Definition 2 ([3]). *The master automaton over an alphabet Σ is a tuple $M = (Q_M, \Sigma, \delta_M, F_M)$ where*

– *Q_M is the set of all weakly acyclic languages over Σ;*
– *$\delta_M : Q_M \times \Sigma \to Q_M$ with $\delta_M(L, a) = L^a$ for every $q \in Q_M$ and $a \in \Sigma$;*
– *$L \in F_M \iff \epsilon \in L$.*

As depicted formally in Definition 2, the master automaton is a DFA without an initial state but with an infinite number of states, where each state is representing a distinct weakly acyclic language. The transitions δ_M are described using the notion of the residual language with respect to a letter (Chapter 2).

Given a state L of the master automaton characterizing the weakly acyclic language L, the language recognized from that state is L [3, Proposition 8]. This means that a DFA for any weakly acyclic language L exists as a finite subautomaton in the master automaton. Formally, that DFA A_L is the tuple $(Q_L, \Sigma, \delta_L, q_{0L}, F_L)$ where

– Q_L is the set of states of the master automaton reachable from state L;
– q_{0L} is the state L;
– δ_L is the projection of δ_M onto Q_L;
– F_L is the intersection of F_M and Q_L.

Furthermore, for every weakly acyclic language L the automaton A_L is also the unique minimal DFA for recognizing L [3, Proposition 9].

We can define the relation $\preceq$ on the set of all weakly acyclic languages Q_M where $L_1 \preceq L_2$ if $L_1 = L_2^w$ for some word w. By the definition of weakly acyclic DFAs $\preceq$ must be a partial order. The minimal elements of the order satisfy $L = L^a$ for every $a \in \Sigma$, which are only $\emptyset$ and Σ^*.

4.2 Table of Nodes

Using the idea of the master automaton, we can construct a data structure to store a finite set of weakly acyclic DFAs, which we call the *Table of Nodes.* For the languages $L_1, L_2, \dots, L_N$ we take their corresponding states in the master automaton and all their reachable successor states and put them as nodes into the table. Every node corresponds to a weakly acyclic language and can be mapped to a state in the master automaton.

As the transitions of the master automaton are defined using residual languages, we can describe each node uniquely by its direct successor nodes for every letter and if it is a final state [3].

For an alphabet $\Sigma = \{a_1, a_2, \dots, a_n\}$ we define a *node* as a tuple (q, s, b) where

- q is the *node identifier*;
- $s = [q_1, q_2, \dots, q_n]$ is the *successor array* of q where q_i is the identifier for the node of the residual language of q with respect to the letter a_i;
- $b \in \{0, 1\}$ is the *final flag* and tells if q is an accepting state (1) or not (0).

For instance, the node for the language $\emptyset$ is $(q_\emptyset, [q_\emptyset, \dots, q_\emptyset], 0)$ and the language of the empty word ϵ is $(q_\epsilon, [q_\emptyset, \dots, q_\emptyset], 1)$.

The Table of Nodes stores a collection of these nodes and contains a bidirectional mapping between the node identifier and the node's pair of successor array and final flag. Concretely, with T as the table, $T[q]$ returns the pair of successor array and final flag (s, b) in one direction, and $T[(s, b)]$ gives back the identifier q of the node in the other direction.

An example for a table for the alphabet $\Sigma = \{\boldsymbol{a}, \boldsymbol{b}\}$ is given in Table 4.1. The graphical view containing multiple automata for this table is shown next to it in Figure 4.1. Every node uniquely represents a weakly acyclic language, for instance, q_4 stands for $L(\boldsymbol{b^*})$, while q_3 for $L(\boldsymbol{ab} + \boldsymbol{ba})$. Therefore, the successor of q_3 for letter a has to characterize $L(\boldsymbol{b})$, which q_1 does indeed.

We define the operations succ and final to access node elements by the node identifier for future algorithms. Given a node identifier q and a letter $a_i \in \Sigma$, $\mathsf{succ}(q, a_i)$ returns the successor element q_i for letter a_i in the successor array s of the node for q. The final flag b of the node identified by q is returned by $\mathsf{final}(q)$.

The following sections are dedicated to introducing and discussing important procedures defined on the Table Of Nodes.

4.3 Creation of Nodes into the Table

This section introduces operations on the Table of Nodes with the purpose to add new nodes. make is responsible for the creation of new nodes and is the only operation which can directly add new nodes to the table. create uses make to create nodes for a restricted weakly acyclic language given as input.

node id	successor array s alphabet order $[\boldsymbol{a}, \boldsymbol{b}]$	final flag b
$q_\emptyset$	$[q_\emptyset, q_\emptyset]$	0
q_ϵ	$[q_\emptyset, q_\emptyset]$	1
q_1	$[q_\emptyset, q_\epsilon]$	0
q_2	$[q_\epsilon, q_\epsilon]$	0
q_3	$[q_1, q_2]$	0
q_4	$[q_\emptyset, q_4]$	1
q_5	$[q_\emptyset, q_3]$	0

Table 4.1: Example for Table of Nodes

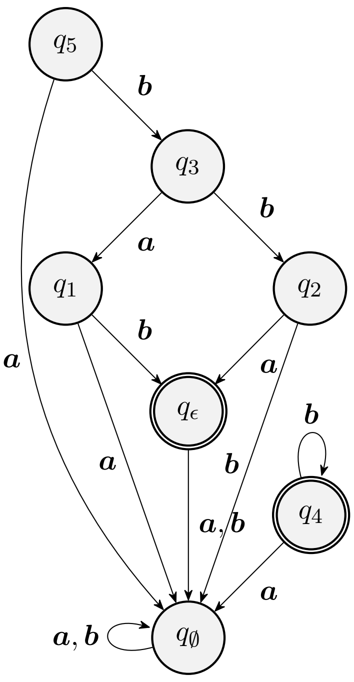


Figure 4.1: Graphical Representation

4.3.1 make - Adding Nodes to the Table

Naturally, the data structure needs an operation for adding new nodes. We define a procedure make, which takes a successor array s and a final flag b as input.

An intuitive procedure is presented in Algorithm 1. The operation is defined on an existing Table of Nodes T. The require statement expresses the precondition that all successor nodes in s must already exist in the table. In the first two lines, we check if a node with the same successors and final value already exists. If there is this node already in T, we return its identifier. If it does not exist, we create a new id for the new node in line 4 and add it into the table T. Note that in addition to the mapping from id to node in line 5, the reverse direction also needs to be inserted into the table in line 6. The new identifier is finally returned.

However, this straight-forward algorithm for make is not completely correct. If we imagine adding a new node with self-loops for every letter, the identifier of this new node is unknown at this point in time. Therefore, we are not able to construct the successor array s correctly. In order to address this issue, we introduce a special identifier SELF, which indicates that the successor is the current node itself. This special id is only used for the creation of a new node in make and will not appear in later operations with the node. For creating the node for $\emptyset$, we then give make the arguments $s = [\text{SELF}, \ldots, \text{SELF}]$ and $b = 0$.

Using this treatment for self-loops, we can derive a more complicated but correct version of make. The major problem with the introduction of SELF is its interchangeability with

Algorithm 1 First Version make (wrong)

```
Input: $s = [q_1, \dots, q_n],\ b \in \{0,1\}$
Require: $\forall q_i \in s : q_i \in T$
1: if $\exists q = T[(s,b)]$ then
2:     return $q$
3: else
4:     $q_{\text{new}} \leftarrow \mathsf{newId}()$
5:     $T[q_{\text{new}}] \leftarrow (s,b)$
6:     $T[(s,b)] \leftarrow q_{\text{new}}$
7:     return $q_{\text{new}}$
```

the node identifier. Imagine the table already contains the node q_1 with $s = [q_1, q_1, q_2, q_3]$. If we create a node with $s = [\text{SELF}, \text{SELF}, q_2, q_3]$ and the same b as q_1, the existing q_1 represents this node already and should be returned. But also for $s = [q_1, \text{SELF}, q_2, q_3]$, $s = [\text{SELF}, q_1, q_2, q_3]$ and $s = [q_1, q_1, q_2, q_3]$ the same holds. In fact, all possible combinations between the node id and SELF characterize the same successor array. In order for the lookup for existing nodes to work correctly, we thus have to add all of these combinations during the creation of q_1.

This is the main change in the next make procedure in Algorithm 2. The first part with the lookup for an existing node stays the same as in Algorithm 1. But if the node does not exist yet, we add all possible combinations, denoted by $\pi(s)$, in lines 5 and 6. For the mapping from identifier to s and b, the successor array without SELFs is used. This is performed by line 7, where $s_{[\text{SELF}/q_{\text{new}}]}$ denotes the array s, with the only difference that each SELF is substituted by q_{new}.

Algorithm 2 Second Version make

```
Input: $s = [q_1, \dots, q_n],\ b \in \{0,1\}$
1: if $\exists q = T[(s,b)]$ then
2:     return $q$
3: else
4:     $q_{\text{new}} \leftarrow \mathsf{newId}()$
5:     for each $s' \in \pi(s)$ do
6:         $T[(s',b)] \leftarrow q_{\text{new}}$
7:     $s \leftarrow s_{[\text{SELF}/q_{\text{new}}]}$
8:     $T[q_{\text{new}}] \leftarrow s$
9:     return $q_{\text{new}}$
```

This adapted procedure is much more expensive than the simple one before. Given a successor array with n self-loops, there are $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ many ways of combining SELF and the actual node id. Thus, the number of permutations are exponential in the number of SELFs in the successor array. By inserting all of them, not only the time efficiency of make suffers, but also the table containing the mappings might grow very large.

To avoid some of these issues, another approach for make is portrayed in Algorithm 3. The insight is that besides the two cases where the successor array only contains either SELF or the actual identifier and no combination of both, in all other cases, the only possible nodes to consider for equality are the non-SELF identifiers existent in the successor array. Therefore, when adding a new node, we only store those two cases for the lookup. This happens in line 8 with the only SELF case, and then in line 10 with the only real identifier case, after we have replaced every SELF with the correct new id in line 9. If nothing has been found in the lookup, the loop from line 3 to 6 iterates over all non-SELF ids q_i present in s and inspects if these nodes are identical to the currently being constructed node. This happens by replacing all SELF ids with q_i into the array s_i, and examining the equivalence of (s_i, b) and the successors and final flag of q_i in line 5 and 6.

Algorithm 3 Third Version make

```
Input: $s = [q_1, \dots, q_n],\ b \in \{0, 1\}$
 1: if $\exists q = T[(s, b)]$ then
 2:     return $q$
 3: for each $q_i \in s : q_i \neq \text{SELF}$ do
 4:     $s_i \leftarrow s_{[\text{SELF}/q_i]}$
 5:     if $T[q_i] = (s_i, b)$ then
 6:         return $q_i$
 7: $q_{\text{new}} \leftarrow \mathsf{newId}()$
 8: $T[(s, b)] \leftarrow q_{\text{new}}$
 9: $s \leftarrow s_{[\text{SELF}/q_{\text{new}}]}$
10: $T[(s, b)] \leftarrow q_{\text{new}}$
11: $T[q_{\text{new}}] \leftarrow (s, b)$
12: return $q_{\text{new}}$
```

Although Algorithm 3 returns the correct node, it is inefficient to iterate over all successors in cases without any SELF present in s. Therefore, a real implementation would first examine, if a SELF exists in s. If there is none, we can simply perform the intuitive make from Algorithm 1.

Algorithm 3 does not have a very fast lookup compared to Algorithm 2, as all the non-SELF nodes need to be iterated over. Especially in cases, where there are very few SELFs and lot of different identifiers, this approach can become more inefficient compared to Algorithm 2. However, the exponential number of permutations in Algorithm 2 is replaced here with a procedure linear in the length of the successor array s.

For our application, a variation of Algorithm 2 has been implemented. The alphabet we are using for our application is very small containing only 3 letters. Furthermore, the table we are constructing will only contain a maximum of one SELF in any successor array. Therefore, even with the exponential approach, only two insertions into table need to be performed, one with only SELFs instead of the real node id and one the other way around.

4.3.2 create - Adding Nodes for a Specific Language

The operation `create` allows to create nodes for a weakly acyclic language given a string str. The string only contains letters of the alphabet Σ or the Kleene star `*`. `create` processes the string character by character and recursively calls itself with shorter substrings.

For instance, with $str =$ `ab*c*d*a` as input `create` constructs the node for $L(\boldsymbol{a}\boldsymbol{b}^*\boldsymbol{c}^*\boldsymbol{d}^*\boldsymbol{a})$ and returns its identifier. The recursive calls for this input are shown in Table 4.2. The first column shows the `create` call with a string as argument. The second and third column show the successor array s and final flag b we construct for that input string. We use this table to describe how `create` works.

create call	constructed successor array s alphabet order $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}]$	final flag b
create(ab*c*d*a)	[create(b*c*d*a), $q_\emptyset, q_\emptyset, q_\emptyset$]	0
create(b*c*d*a)	[create(), SELF, create(c*d*a), create(d*a)]	0
create(c*d*a)	[create(), $q_\emptyset$, SELF, create(d*a)]	0
create(d*a)	[create(), $q_\emptyset, q_\emptyset$, SELF]	0
create()	[$q_\emptyset, q_\emptyset, q_\emptyset, q_\emptyset$]	1

Table 4.2: Example for create

First, the operation distinguishes between two cases. In the first one, the input string starts with a letter not followed by the Kleene star `*` character. This corresponds to the first row of the table, where there is no `*` directly after the first character `a`. In this scenario, we create a node which only has one successor for the letter $\boldsymbol{a}$ and all other successors in the successor array s are set to $q_\emptyset$. For the successor with respect to $\boldsymbol{a}$ we make a recursive call with the substring starting from the character after `a`.

The second case is about having a letter followed by `*` at the start of the input string. Obviously, the node we construct needs a self-loop for this letter. This can be seen in the middle three rows of Table 4.2, where the SELF identifier is put into the corresponding position. Furthermore, if there are more letter-star pairs following, we need to identify the first letter without the `*` immediately afterwards. In the call depicted in the second row of the table, we need to iterate over `c*` and `d*` before we find the letter `a`. Because $\boldsymbol{c}^*$ and $\boldsymbol{d}^*$ could both potentially be empty, reading in an $\boldsymbol{a}$ from the current node is also possible. Therefore, for the successor position of $\boldsymbol{a}$ we make another `create` call with the substring after `a`, which in this case is the empty string. For the successor node for $\boldsymbol{d}$, again $\boldsymbol{c}^*$ might be empty, and thus we also make a `create` call with the substring after `c*`. Analogous to this, for the successor with respect to letter ***c*** we call `create` with the substring after `b*`.

However, there are some input strings for which the operation does not process correctly. Consider the input `a*a`. `create` will start constructing a self-loop for the first state, but then it will replace the self-loop by a transition with letter ***a***.

Concretely, for all strings where a character `a` followed by `*` occurs and the first character not followed by `*` thereafter is exactly `a`, our procedure does not create the correct language. This special case is not handled, as it will not be a problem for the future use of `create` and would slow down the otherwise simple procedure.

4.4 Recursive Algorithms on the Table

The acyclic structure of the partial order $\preceq$ from Section 4.1 with minimal elements $\emptyset$ and Σ^* allows us to define operations on the table recursively with $q_\emptyset$ and q_{Σ^*} as the recursion base cases.

An example for such a recursive algorithm is given with the `union` operation in Algorithm 4. The input q_1 and q_2 are two existing nodes in the table and the output is the node of their union, which is added into the table during the procedure. A cache stores already computed union results.

In the first lines, a lookup in the cache is performed. If nothing is found, the boundary

Algorithm 4 Union of Two Nodes

```
1:  procedure union($q_1, q_2$)
2:      if cache[$\{q_1, q_2\}$] then
3:          return cache[$\{q_1, q_2\}$]
4:      else if $q_1 = q_{\Sigma^*}$ or $q_2 = q_{\Sigma^*}$ then
5:          return $q_{\Sigma^*}$
6:      else if $q_1 = q_\emptyset$ then
7:          return $q_2$
8:      else if $q_2 = q_\emptyset$ then
9:          return $q_1$
10:     $b \leftarrow (\mathsf{final}(q_1) \vee \mathsf{final}(q_2))$
11:     cache[$\{q_1, q_2\}$] $\leftarrow$ SELF
12:     $s \leftarrow [q_\emptyset, \dots, q_\emptyset]$
13:     for each $a_i \in \Sigma$ do
14:         $s[i] \leftarrow \mathsf{union}(\mathsf{succ}(q_1, a_i), \mathsf{succ}(q_2, a_i))$
15:     $q_{\text{union}} \leftarrow T.\mathsf{make}(s, b)$
16:     cache[$\{q_1, q_2\}$] $\leftarrow q_{\text{union}}$
17:     return $q_{\text{union}}$
```

cases with the minimal elements are examined. The union of a node with Σ^* is always Σ^*, which is handled in line 4 and 5. Lines 6 to 9 address the cases with $q_\emptyset$, where the union of a node with $\emptyset$ is the node itself. If this is also not the case, we are constructing the correct successor array and final flag for this node before passing them to make.

The calculation of b is shown in line 10. If either one of q_1 and q_2 is an accepting node, then their union is accepting too. For the derivation of the successor nodes, the residual of the union of q_1 and q_2 with respect to a letter is required. We use the equality $(L_1 \cup L_2)^a = L_1^a \cup L_2^a$. For every letter a_i in Σ, we recursively perform union with the successor nodes of both q_1 and q_2 with respect to a_i in line 14.

After the construction of s and b, we give them to make and write the id given back by make into the cache in line 15 before returning it.

Similar to the make procedure, the possibility of self-loops remain a small obstacle for union. If there is a letter a where both $q_1 = \mathsf{succ}(q_1, a)$ and $q_2 = \mathsf{succ}(q_2, a)$, the successor array of the union also has a self-loop for letter a. Consequently, the special id SELF has to be written into the position of letter a in s. We can achieve this quite elegantly by writing SELF into the cache for $\{q_1, q_2\}$ in line 11 before the computation of s. If $\mathsf{union}(q_1, q_2)$ is called in line 14, line 2 will return the SELF identifier, which has been written into the cache on the previous recursion level. We only need to overwrite the SELF in the cache with the correct identifier afterwards in line 16.

Other recursive methods on the table, like intersection for the computation of the intersection of two nodes, have a very similar structure. There is always a cache for storing already calculated results. The boundary cases in the beginning involve $q_\emptyset$ and q_{Σ^*}, the nodes in the successor array are iterated over, and a recursive call is made for each successor node.

4.5 pre - An Algorithm for Computing the Preimage of a Transducer

In this section we present the important operation pre, which computes the preimage of a transducer given a weakly acyclic DFA. We begin by introducing the Pre language, which is a formal description of that preimage, before we sketch an intuitive but wrong algorithm for pre computing this language. We show where the problem lies and add appropriate measures to achieve the correct final version of pre.

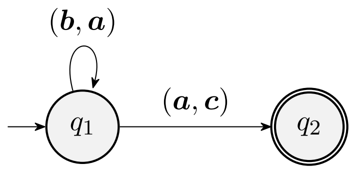
4.5.1 The Pre Language

Recall that a transducer $\mathcal{T}$ is an NFA over $\Sigma \times \Sigma$ and defines a mapping from words of one regular language to words of another regular language. The transitions $\delta_\mathcal{T}$ can be described by the tuple $(p, (a_1, a_2), p')$, where the transducer state p transitions to state p' when reading the pair (a_1, a_2).

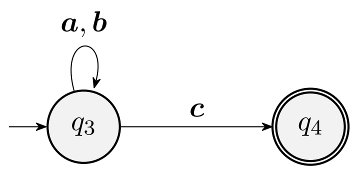
Definition 3 ([3])**.** *$Pre_{\mathcal{T}}(A) = \{w \in \Sigma^* : \exists v \in L(A)$ s.t. $(w, v) \in L(\mathcal{T})\}$ is the* Pre language *of a transducer $\mathcal{T} = (Q_{\mathcal{T}}, \Sigma, q_{0\mathcal{T}}, \delta_{\mathcal{T}}, F_{\mathcal{T}})$ with respect to a DFA $A = (Q_A, \Sigma, q_{0A}, \delta_A, F_A)$.*

A word w is in the Pre language if the transducer maps w to a word v which is also accepted by automaton A. Therefore, $\mathrm{Pre}_{\mathcal{T}}(A)$ is the preimage of transducer $\mathcal{T}$ corresponding to the image $L(A)$.

A concrete example is depicted in Figure 4.2 with a transducer on the left and a DFA on the right. The trap state is left out in the figure of the DFA. The language of A in this case is $L((\boldsymbol{a} + \boldsymbol{b})^*\boldsymbol{c})$. The transducer $\mathcal{T}$ in Figure 4.2a maps words of the form $\boldsymbol{b}^k\boldsymbol{a}$ to $\boldsymbol{a}^k\boldsymbol{c}$ with $k \in \mathbb{N}$. Since here the complete image of the transducer $L(\boldsymbol{a}^*\boldsymbol{c})$ is contained in $L(A)$, $\mathrm{Pre}_{\mathcal{T}}(A)$ is the language $L(\boldsymbol{b}^*\boldsymbol{a})$.



(a) Transducer $\mathcal{T}$



(b) DFA A

Figure 4.2: Example of Pre Language

Lemma 2 ([3])**.** *The Pre language $Pre_{\mathcal{T}}(A)$ can be described by the NFA $B = (Q_B, \Sigma, q_{0B}, \delta_B, F_B)$ where*

- $Q_B = Q_{\mathcal{T}} \times Q_A$;
- $q_{0B} = (q_{0\mathcal{T}}, q_{0A})$;
- $F_B = F_{\mathcal{T}} \times F_A$;
- $\delta_B((p, q), x) = \{(p', q') : \exists y \in \Sigma$ *such that* $p' \in \delta_{\mathcal{T}}(q, (x, y)), q' = \delta_A(q, y)\}$.

Lemma 2 defines a construction of an NFA B which accepts the Pre language from Definition 3. The structure of B shows similarities with the product construction for the intersection of two DFAs. In fact, B can be viewed precisely as the product of the intersection of the DFA A with the image of the transducer $\mathcal{T}$.

Figure 4.3 shows B for the example in Figure 4.2. Each state in B is a pair of a state in $\mathcal{T}$ and a state in A. The initial state q_{0B} is the pair of initial states of $\mathcal{T}$ and A, in Figure 4.3 this is the pair (q_1, q_3). The final states are the pairs of final states of both $\mathcal{T}$ and A. In Figure 4.3, the state (q_2, q_4) is final since q_2 and q_4 are also final in the transducer and the DFA.

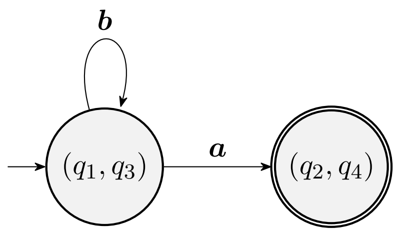


Figure 4.3: NFA B for $\mathrm{Pre}_{\mathcal{T}}(A)$ for Figure 4.2

The transition function δ_B takes a state (p, q), where p is a state of the transducer and q a state of the DFA, and a letter x. The set of states given back are all (p', q') where a letter y with the following conditions exists:

– In the transitions of $\mathcal{T}$ there exists the transition $p \xrightarrow{(x,y)} p'$.
– In the transitions of A there exists the transition $q \xrightarrow{y} q'$.

Taking Figure 4.3 as an example, there exists the transition from $(p, q) = (q_1, q_3)$ to $(p', q') = (q_2, q_4)$ with letter $x = \boldsymbol{a}$ in B, because there is $y = \boldsymbol{c}$ such that:

– In the transitions of $\mathcal{T}$ there exists the transition $q_1 \xrightarrow{(\boldsymbol{a},\boldsymbol{c})} q_2$ (Figure 4.2a).
– In the transitions of A there exists the transition $q_3 \xrightarrow{\boldsymbol{c}} q_4$ (Figure 4.4b).

Using this construction, the complete NFA B in Figure 4.3 characterizes the language $L(\boldsymbol{b}^*\boldsymbol{a})$ as expected.

As we operate with weakly acyclic languages on the Table Of Nodes, an important question arises: Are weakly acyclic languages also closed under the Pre language? Specifically, given a weakly acyclic A and a weakly acyclic $\mathcal{T}$, we wonder if $\mathrm{Pre}_{\mathcal{T}}(A)$ also remains weakly acyclic. Note here that we call a transducer weakly acyclic if the underlying NFA is weakly acyclic. Blondin et al. have shown that this is not the case in general. They introduce an additional syntactically defined property for transducers which guarantees the weakly acyclicity for the result of $\mathrm{Pre}_{\mathcal{T}}(A)$ [3].

4.5.2 A Simple but Wrong Algorithm

We aim to incorporate the calculation of the Pre language into the Table Of Nodes with the operation pre. For a transducer $\mathcal{T}$ and a node with id q, pre returns the node for $\mathrm{Pre}_{\mathcal{T}}(q)$, which either has already been present in the table or has been added during this operation.

We already encounter two impediments with the integration into the table. The first one, already mentioned in the last chapter, is about the result of the pre operation not always being weakly acyclic. This is a substantial problem as the table has been created for storing weakly acyclic languages, and the operations on the table take advantage of the properties of this specific class of languages. The second problem reflects that the result automaton B in

Lemma 2 is an NFA and might be nondeterministic. The table only stores weakly acyclic DFAs, so another step of determinizing B into an equivalent DFA C is required.

For the first impediment, we are not introducing any restrictions on the transducer to ensure that the resulting language is weakly acyclic. We guarantee a correct result of the operation if the result is weakly acyclic. However, if this is not fulfilled, the algorithm will behave undefined.

To address the issue of nondeterminism, we use the idea behind the powerset construction, where the states of the DFA represent a set of states from the NFA.

Using these ideas, we develop the following conceptual approach for pre with the precondition that $\mathrm{Pre}_{\mathcal{T}}(A)$ is a weakly acyclic language:

- Construct NFA B for $\mathrm{Pre}_{\mathcal{T}}(A)$ according to Lemma 2.
- Determinize NFA B into the DFA C using the power set construction.

We first present a more simple but erroneous algorithm of pre implementing this approach, which is sketched in Algorithm 5. We write Q_T for all nodes and F_T for all accepting nodes in the table T. Analogous to the last section, the transducer $\mathcal{T}$ is of the form $(Q_{\mathcal{T}}, \Sigma, q_{0\mathcal{T}}, \delta_{\mathcal{T}}, F_{\mathcal{T}})$.

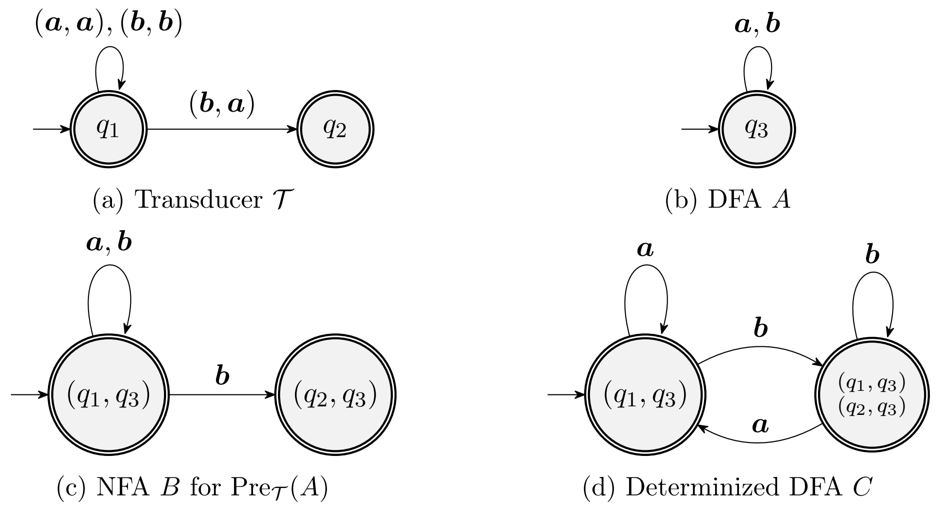
Algorithm 5 First Version pre (wrong)

```
Input: $S \subseteq Q_{\mathcal{T}} \times Q_T, \mathcal{T}$
1:  procedure pre($S, \mathcal{T}$)
2:      if cache[$S$] then
3:          return cache[$S$]                                        ▷ cache lookup
4:      $b \leftarrow (S \cap (F_{\mathcal{T}} \times F_T) \neq \emptyset)$          ▷ $F_B$ from Lemma 2
5:      $s \leftarrow [q_\emptyset, \dots, q_\emptyset]$
6:      for each $a_i \in \Sigma$ do
7:          $S' \leftarrow \emptyset$
8:          for each $(p, q) \in S$ do
9:              $S' \leftarrow S' \cup \{(p', q') : \exists y \in \Sigma$ such that $p' \in \delta_{\mathcal{T}}(q, (a_i, y)), q' = \mathsf{succ}(q, a_i)\}$
10:         if $S = S'$ then
11:             $s[i] \leftarrow$ SELF                                   ▷ self-loop case
12:         else
13:             $s[i] \leftarrow \mathsf{pre}(S', \mathcal{T})$                  ▷ recursive call with $S'$
14:     $q_{\mathrm{pre}} \leftarrow T.\mathsf{make}(s, b)$                          ▷ create node
15:     cache[$S$] $\leftarrow q_{\mathrm{pre}}$                              ▷ save result in cache
16:     return $q_{\mathrm{pre}}$
```

By operating with sets of states from the NFA B throughout the whole algorithm, we immediately generate the corresponding states of the determinized DFA C. Therefore, the first argument S of pre represents a single state in the DFA C and is composed of states from NFA B. Each state in B is a pair containing a state of the transducer ($p \in Q_{\mathcal{T}}$) in the first position and a state of the table ($q \in Q_T$) in the second position, as described in Lemma 2.

In order to construct the right node for the Pre language, we compute the correct final flag b in line 4 and successor array s in the loop from line 6 to 13. After the inner loop from line 8 to 9, the variable S' represents the successor state in DFA C for S with respect to the letter a_i. This happens by accumulating all $\delta_B((p,q),a_i)$ from Lemma 2 into S' in line 9. Recall that $\mathsf{succ}(q, a_i)$ returns the node for the successor language with respect to the letter a_i, which agrees with $\delta_A(q, a_i)$ from Lemma 2. After the loop in line 14, we pass the constructed s and b to the make operation, which creates the node into the table.

This algorithm is not functioning correctly for all cases where the Pre language is weakly acyclic. Consider the example given in Figure 4.4 with $\Sigma = \{\boldsymbol{a}, \boldsymbol{b}\}$. The NFA B for the Pre language in Figure 4.4c is portraying the language $L((\boldsymbol{a}+\boldsymbol{b})^*)$, which is weakly acyclic since its minimal automaton is only one state with a self-loop for $\boldsymbol{a}$ and $\boldsymbol{b}$. However, after determinizing B, the result automaton C shown in Figure 4.4d is not weakly acyclic and contains a cycle. This poses a serious problem for the pre procedure we have established in Algorithm 5.


$(\boldsymbol{a},\boldsymbol{a}),(\boldsymbol{b},\boldsymbol{b})$
$(\boldsymbol{b},\boldsymbol{a})$
q_1
q_2
$\boldsymbol{a},\boldsymbol{b}$
q_3
$\boldsymbol{a},\boldsymbol{b}$
$\boldsymbol{b}$
(q_1,q_3)
(q_2,q_3)
$\boldsymbol{a}$
$\boldsymbol{b}$
$\boldsymbol{b}$
$\boldsymbol{a}$
(q_1,q_3)
(q_1,q_3) (q_2,q_3)
(a) Transducer $\mathcal{T}$

(b) DFA A

(c) NFA B for $\mathrm{Pre}_{\mathcal{T}}(A)$

(d) Determinized DFA C

Figure 4.4: Counter Example for Algorithm 8

Imagine the example from Figure 4.4 as input for Algorithm 5. A recursion call tree for this concrete case is drawn in Figure 4.5.

The algorithm starts with the input $S = \{(q_1, q_3)\}$, which is the initial node in Figure 4.4d and the root of the recursion tree. In the first iteration of the loop for letter $\boldsymbol{a}$, starting from line 6, the set S' is the same as S, which can also be recognized in Figure 4.4d with the self-loop in state (q_1, q_3). Therefore, the condition in line 10 evaluates to true and no recursive call is made in this iteration. In the tree, this case is depicted in the left child of

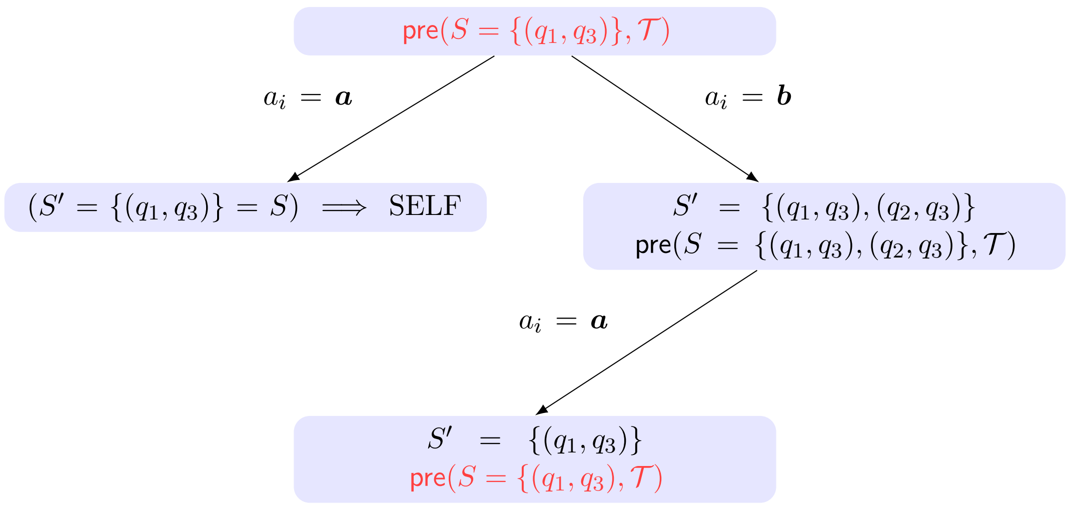

pre($S = \{(q_1, q_3)\}, \mathcal{T}$)
$a_i = \boldsymbol{a}$
$a_i = \boldsymbol{b}$
$(S' = \{(q_1, q_3)\} = S) \implies$ SELF
$S' = \{(q_1, q_3), (q_2, q_3)\}$
pre($S = \{(q_1, q_3), (q_2, q_3)\}, \mathcal{T}$)
$a_i = \boldsymbol{a}$
$S' = \{(q_1, q_3)\}$
pre($S = \{(q_1, q_3), \mathcal{T}$)

Figure 4.5: Recursion Tree for Counter Example from Figure 4.4

the root. In the second iteration with letter $\boldsymbol{b}$, pre is called with $S' = \{(q_1, q_3), (q_2, q_3)\}$ in line 13, which is the right child of the root in the recursion tree.

Now we are one recursion level down and the current input $S = \{(q_1, q_3), (q_2, q_3)\}$. Again, we iterate over the successors starting with letter $\boldsymbol{a}$ and compute $S' = \{(q_1, q_3)\}$. At this point, pre with argument $S' = \{(q_1, q_3)\}$ is called in line 13. This recursive call is illustrated in the bottom node of the recursion tree. However, we can notice that this is exactly the same pre function call we performed in the beginning in the root, both marked in red in Figure 4.5. Therefore, for this example Algorithm 5 will run into an infinite loop without terminating, even though the Pre language is a weakly acyclic language.

4.5.3 A High-Level Algorithm

To confront the issue of possible cycles in the determinized DFA, we shift the perspective from the concrete implementation of the pre operation back to a more abstract level on the correct construction of a weakly acyclic DFA accepting the Pre language. We can then adapt Algorithm 5 to also construct that DFA into the Table of Nodes.

Algorithm 6 describes a procedure for constructing the minimal weakly acyclic DFA $\widetilde{C}$, which recognizes $\text{Pre}_{\mathcal{T}}(A)$. The precondition states that $\text{Pre}_{\mathcal{T}}(A)$ is a weakly acyclic language. Again like in the last subsection, the NFA B is formed and determinized into C. As we have seen with the example in Figure 4.4, this might produce a not weakly acyclic C. Therefore, in the loop from line 3 to 4 all cycles beyond self-loops are removed by collapsing the states part of the cycle. Here, collapsing $q_1, q_2, \ldots, q_k$ into one state means removing all nodes in the DFA except q_1 and redirecting any incoming transitions of $q_2, \ldots, q_k$ to q_1. Line 6 performs a minimization on C returning $\widetilde{C}$. With Lemma 3 we show that this procedure yields a correct minimal weakly acyclic DFA for the language $\text{Pre}_{\mathcal{T}}(A)$.

Algorithm 6 Construction of Weakly Acyclic DFA for Pre Language

```
Input: transducer $\mathcal{T}$, weakly acyclic DFA $A$
Output: minimal weakly acyclic DFA $\widetilde{C}$
Require: $\mathrm{Pre}_{\mathcal{T}}(A)$ is weakly acyclic
1: construct NFA $B$ for $\mathrm{Pre}_{\mathcal{T}}(A)$ according to Lemma 2
2: determinize NFA $B$ into the DFA $C$ using the power set construction
3: while there is a cycle $q_1, q_2, \dots, q_k$ with $q_1 \neq q_2$ in $C$ do
4:     collapse states $q_1, q_2, \dots, q_k$ into single state
5: minimize $C$ into minimal automaton $\widetilde{C}$
```

Lemma 3. *After execution of Algorithm 6, $\widetilde{C}$ is a minimal weakly acyclic automaton for the Pre language $Pre_{\mathcal{T}}(A)$.*

Proof. The fact that $\widetilde{C}$ is weakly acyclic is intuitive since all cycles beyond self-loops have been removed by the loop in line 3 and 4 of Algorithm 6, which agrees with the definition a weakly acyclic DFAs. Furthermore, the minimization step in line 5 guarantees the minimality of $\widetilde{C}$.

We aim to show that $L(\widetilde{C}) = \mathrm{Pre}_{\mathcal{T}}(A)$.

Let C_0 denote C before entering the loop, C_j be C after the j-th iteration of the loop and C_n be C after the last iteration of the loop.

With Lemma 2 we have $L(B) = \mathrm{Pre}_{\mathcal{T}}(A)$ after line 1. For the DFA C_0 constructed by the power set construction in line 4, it holds that $L(C_0) = L(B) = \mathrm{Pre}_{\mathcal{T}}(A)$. Moreover $L(C_n) = L(\widetilde{C})$ because the minimization step in line 5 does not change the language recognized by the automaton.

Therefore, to show $L(\widetilde{C}) = \mathrm{Pre}_{\mathcal{T}}(A)$ we only need to prove that the loop iterations do not change the language of the automaton: $L(C_j) = L(C_{j+1})\ \forall\ 0 \leq j < n$.

Let $\widehat{C}$ denote the unique minimal automaton of C_0. Each state in $\widehat{C}$ recognizes a distinct language and trivially $L(\widehat{C}) = L(C_0)$. Recall that $L_A(q)$ denotes the language of state p in automaton A.

Assume for contradiction there exists a minimal $i < n$ such that $L(C_i) \neq L(C_{i+1})$. This means in the $i+1$-th iteration of the loop in Algorithm 6 non-equivalent states have been collapsed. Specifically, there exist two distinct states p, q, which are part of a cycle in C_i and have been collapsed into one state of C_{i+1}, but $L_{C_i}(p) \neq L_{C_i}(q)$.

As p, q form a cycle in C_i, there exist non-empty words $v, w \in \Sigma^*$ such that $\delta_{C_i}(p, v) = q$ and $\delta_{C_i}(q, w) = p$.

As we chose i to be minimal, $L(C_i) = L(C_0) = L(\widehat{C})$, which means $\widehat{C}$ is also the unique minimal automaton for C_i. Since $L_{C_i}(p) \neq L_{C_i}(q)$, there exist distinct states p', q' in $\widehat{C}$ such that $L_{C_i}(p) = L_{\widehat{C}}(p')$ and $L_{C_i}(q) = L_{\widehat{C}}(q')$.

Then it holds that

- $\delta_{\widehat{C}}(p', v) = q'$ because $L_{\widehat{C}}(p')^v = L_{C_i}(p)^v = L_{C_i}(q) = L_{\widehat{C}}(q')$;
- $\delta_{\widehat{C}}(q', w) = p'$ because $L_{\widehat{C}}(q')^w = L_{C_i}(q)^w = L_{C_i}(p) = L_{\widehat{C}}(p')$.

This shows there also exists a cycle in $\widehat{C}$ containing the distinct states p' and q', which means $\widehat{C}$ is not a weakly acyclic DFA. With Lemma 1 the unique minimal DFA for a weakly acyclic language is also weakly acyclic, which shows that $L(\widehat{C})$ can not be weakly acyclic.

However, $L(\widehat{C}) = L(C_0) = \text{Pre}_{\mathcal{T}}(A)$ and $\text{Pre}_{\mathcal{T}}(A)$ is weakly acyclic by the precondition in Algorithm 6. Therefore, the assumption has been wrong and $L(C_j) = L(C_{j+1}) \; \forall \; 0 \leq j < n$ holds, which proves that $L(\widetilde{C}) = \text{Pre}_{\mathcal{T}}(A)$.

□

4.5.4 Implementing the Final Version

Finally, we are integrating the idea from Algorithm 6 into the pre procedure of the Table of Nodes. The previous Algorithm 5 already covered the first two lines from Algorithm 6. What needs to be added at this point is the handling of possible cycles. Algorithm 7 shows the adapted pre function, which is an implementation of the abstract Algorithm 6. The blue colored lines demonstrate what has been extended to the first version in Algorithm 5.

We add the special identifier INVALID, which is used to recognize the existence of a cycle. In line 4 of Algorithm 7 we write INVALID into the cache and only when the node has been added into the table, this INVALID is overwritten with the correct id q_{pre} in line 23. Therefore, during the construction of the successor array s of the current S, this special identifier stays in the cache. If we encounter INVALID in the cache for S' in line 13, the successors for S' are currently still being constructed, which means a future successor of S' is again S'. Furthermore, this future successor is not a direct successor of S' as the self-loop case has already been handled in lines 11 and 12 before.

With Algorithm 6, we can use the approach of collapsing to handle the cycles recognized with INVALID. When we come across INVALID for S' in line 13, we move up all recursion levels until we are in the level where the node for that S' is constructed, and we collapse this cycle over multiple layers into a self-loop. Concretely, the recognition of a cycle happens in line 13, after which we return a special abort pair in line 14. The first element of this pair is the label ABORT, indicating we are in the process of wandering up the recursion layers, and the second element is S', which is showing where the cycle started from. In line 17, we handle the occurrence of the abort pair. If we have reached the correct level where the cycle initially began, checked in line 18, we set this successor $s[i]$ to a self-loop. Otherwise, we are still in a recursion level in between and we just pass on in line 21 what has been returned in line 16.

The last line of Algorithm 6 yields the minimal automaton for the Pre language, which is necessary for the table, as it only stores minimal weakly acyclic DFAs. This step is implicitly resolved by the make operation in line 22 of Algorithm 7. There can not be two nodes in the table characterizing the same language, because their successor array and final flag are

Algorithm 7 Final Version pre

```
1:  procedure pre($S \subseteq Q_\mathcal{T} \times Q_\mathcal{T}, \mathcal{T}$)
2:      if cache[$S$] then
3:          return cache[$S$]
4:      cache[$S$] $\leftarrow$ INVALID
5:      $b \leftarrow (S \cap (F_\mathcal{T} \times F_\mathcal{T}) \neq \emptyset)$
6:      $s \leftarrow [q_\emptyset, \dots, q_\emptyset]$
7:      for each $a_i \in \Sigma$ do
8:          $S' \leftarrow \emptyset$
9:          for each $(p, q) \in S$ do
10:             $S' \leftarrow S' \cup \{(p', q') : p' \in \delta_\mathcal{T}(p, (a_i, b)), q' = \mathsf{succ}(q, a_i)$ for some $b \in \Sigma\}$
11:         if $S = S'$ then
12:             $s[i] \leftarrow$ SELF
13:         else if cache[$S'$] = INVALID then
14:             return (ABORT, $S'$)
15:         else
16:             $s[i] \leftarrow \mathsf{pre}(S', \mathcal{T})$
17:             if $s[i]$ = (ABORT, $X$) then
18:                 if $X = S$ then
19:                     $s[i] \leftarrow$ SELF
20:                 else
21:                     return (ABORT, $X$)
22:     $q_{\text{pre}} \leftarrow T.\mathsf{make}(s, b)$
23:     cache[$S$] $\leftarrow q_{\text{pre}}$
24:     return $q_{\text{pre}}$
```

equivalent and make only creates unique nodes. Therefore, the adapted Algorithm 7 is a correct implementation for the computation of the node for a weakly acyclic Pre language.

For the future use of pre we want to calculate the Pre language of a transducer $\mathcal{T}$ with respect to an existing node q of the table. However, the first argument of pre, as we have defined it, expects the set $S \subseteq Q_\mathcal{T} \times Q_\mathcal{T}$ and not a single node of the table. Therefore, when making the first pre call for the node q and transducer $\mathcal{T}$, we pass the set $S = \{(q_{0\mathcal{T}}, q)\}$, which is exactly the initial state of the automaton for the Pre language according to Lemma 2.

5 The Backwards Reachability Algorithm

The main purpose of the thesis is to use weakly acyclic languages to address the coverability problem in Petri nets. Therefore, the data structure Table of Nodes from Chapter 4 and the operations it provides are used in order to realize the Backwards Reachability Algorithm for solving this decision problem. This chapter begins with revisiting the coverability problem and describing the general procedure of the Backwards Reachability Algorithm before adapting it to integrate the Table of Nodes.

5.1 Abstract Backwards Reachability Algorithm

Given a Petri Net N with start marking M_0, we say that a marking M is coverable in (N, M_0), if there exists a reachable marking M', where $M' \geq M$.

The coverability problem decides for a Petri Net, a start marking M_0, and a target marking M if there is a sequence of transitions leading to a marking M' which covers M. It has been shown that the coverability problem is EXPSPACE-complete [6, 17].

The method used in this thesis for solving this problem is the Backwards Reachability Algorithm. This method iteratively calculates the set of all predecessor markings which can cover M, and verifies if M_0 is part of the set. Formally, we define the following set:

Definition 4. *$pred_N(\mathcal{M}) = \bigcup_{t \in Tr} \{M : \exists M' \in \mathcal{M}\ s.t.\ M \xrightarrow{t} M'\}$ is the* Predecessor set *and contains the set of all markings M which evolve into a marking in $\mathcal{M}$ by firing one transition t of the Petri net $N = (P, Tr, F)$.*

The abstract procedure of the Backwards Reachability Algorithm is presented in Algorithm 8. The input are the Petri net N, the start marking M_0, and the target marking M.

In line 1, we initialize the set $\mathcal{M}$ with the upward closed set with M as the sole minimal element. $\mathcal{M}$ contains all markings M' which cover M.

The rest of the procedure happens in the while loop from line 3 onwards. In each iteration, the predecessors $pred_N(\mathcal{M})$ of all markings in $\mathcal{M}$ are computed in line 5 and inserted into $\mathcal{M}$ in line 6.

In line 7 we check if the start marking M_0 has been added to this accumulating set of predecessors. If it is contained in $\mathcal{M}$, there exists a series of transitions transforming M_0 into a marking, which covers M. This indicates M is coverable in (N, M_0) and we return true.

At the start of each iteration in line 4, $\mathcal{M}_{\text{old}}$ is set to $\mathcal{M}$ before $\mathcal{M}$ is extended with the predecessors. Therefore, the comparison of $\mathcal{M}$ and $\mathcal{M}_0$ in line 9 is confirming if any new

Algorithm 8 Backwards Reachability Algorithm

```
Input: $N, M_0, M$
 1: $\mathcal{M} \leftarrow \{M' : M' \geq M\}$
 2: $\mathcal{M}_{\text{old}} \leftarrow \emptyset$
 3: while true do
 4:     $\mathcal{M}_{\text{old}} \leftarrow \mathcal{M}$
 5:     $\mathcal{M}_{\text{pre}} \leftarrow \mathit{pred}_N(\mathcal{M})$
 6:     $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_{\text{pre}}$
 7:     if $M_0 \in \mathcal{M}$ then
 8:         return true
 9:     if $\mathcal{M} = \mathcal{M}_{\text{old}}$ then
10:         return false
```

elements have been added into $\mathcal{M}$ in line 5. If $\mathcal{M}$ has not changed in this iteration, it also will not change in any future one, indicating that M is not coverable and we return false.

As the abstract Algorithm 8 operates on infinite sets of markings, it is not directly implementable in this form. Furthermore, one might wonder if there exist inputs where the conditions in line 7 and line 9 will never evaluate to true, leading to the algorithm's failure to terminate.

Lemma 4. *During the execution of Algorithm 8 the set $\mathcal{M}$ remains upward closed.*

Proof. Upward closed sets are closed under union and *pred* [9, Lemma 3.2.16]. □

Lemma 5. *Algorithm 8 terminates.*

Proof. Follows immediately from [9, Lemma 3.2.17]. □

As described in Chapter 2, each upward closed set can be described by a finite number of minimal elements. Furthermore, Lemma 4 shows that the sets used in the algorithm always stay upward closed. Lemma 5 demonstrates that the algorithm does terminate. Existing implementations use these properties and express $\mathcal{M}$ with its finite set of minimal markings during the whole procedure.

5.2 Using the Table of Nodes for Backwards Reachability

Algorithm 8 is working with infinite upward closed sets of markings, like $\mathcal{M}$ and $\mathcal{M}_{\text{old}}$. Since automata are characterizing languages, which are a set of words, they are a way to describe these infinite sets in a compact finite representation. For our approach, we want to implement the Backwards Reachability Algorithm with the Table of Nodes from Chapter 4 by modelling the markings of Petri nets with weakly acyclic languages. Figure 5.1 shows the mapping between the infinite sets and the nodes in the table.

$$\begin{tikzcd}
\mathcal{M}_1 \arrow[r, "pred\ +\ \cup"] \arrow[d, leftrightarrow] & \mathcal{M}_2 \arrow[r, "pred\ +\ \cup"] \arrow[d, leftrightarrow] & \mathcal{M}_3 \arrow[r, dashed] \arrow[d, leftrightarrow] & \mathcal{M}_n \arrow[d, leftrightarrow] \\
q_1 \arrow[r, "T.\mathsf{pre}\ +\ T.\mathsf{union}"] & q_2 \arrow[r, "T.\mathsf{pre}\ +\ T.\mathsf{union}"] & q_3 \arrow[r, dashed] & q_n
\end{tikzcd}$$

Figure 5.1: Mapping between Markings and Nodes

In the upper row of the figure, the procedure from Algorithm 8 with infinite sets of markings is shown. $\mathcal{M}_i$ denotes the upward closed set $\mathcal{M}$ in iteration i of the loop starting in line 3 of Algorithm 8, and iteration n marks the last iteration where the procedure terminates. With the calculation of the Predecessor set *pred* and set union $\cup$, we transition from the set $\mathcal{M}_i$ to $\mathcal{M}_{i+1}$.

The lower row shows the implementation of the Backwards Reachability Algorithm making use of the Table of Nodes. We map the infinite sets $\mathcal{M}_i$ to nodes q_i in the table using a specific encoding scheme. The operations *pred* and $\cup$ are performed with equivalent table operations pre and union, which were examined in Chapter 4. Given the node q_i, which is encoding $\mathcal{M}_i$, performing one iteration in the lower row yields the node q_{i+1}, which is exactly the encoding for $\mathcal{M}_{i+1}$.

Recall that the pre procedure has been defined with the important precondition that the Pre language needs to be weakly acyclic. In order for pre in the lower row of Figure 5.1 to function correctly, we need to guarantee that every $\mathcal{M}_i$ is mapped to a weakly acyclic node. With Lemma 4 every $\mathcal{M}_i$ is upward closed. Therefore, we need to design the encoding in a way such that all upward closed sets of markings are mapped to weakly acyclic languages.

Algorithm 9 shows the adjusted Backward Reachability Algorithm. On the left side the implementation with the Table of Nodes is sketched. On the right, the corresponding lines of the previous Algorithm 8 are depicted in gray.

Analogous to Algorithm 8, the input are a Petri net, a start marking, and a target marking. In line 1 the Table of Nodes *table* is initialized with an empty table containing only the nodes $q_\emptyset$, q_{Σ^*} and q_ϵ.

First, the input markings need to be transformed into their particular representation of a node in the table. For this, we introduce new operations encode and encodeUpwardClosed. encode converts one marking to a node, while encodeUpwardClosed converts an upward closed set with one minimal marking into a node into the table. The latter is used in line 4 and corresponds to the initialization of $\mathcal{M}$ on the right side of Algorithm 9.

The rest of the adapted operation remains very close to Algorithm 8. Each gray line on the right from the previous algorithm can be exactly mapped to a line of the adapted operation including the table on the left. The sets $\mathcal{M}, \mathcal{M}_{\text{old}}$ and $\mathcal{M}_{\text{pre}}$ are encoded by the nodes q, q_{old} and q_{pre}. The start marking M_0 corresponds to q_0. As already presented with Figure 5.1,

Algorithm 9 First Version of Backwards Reachability Algorithm with Table of Nodes

```
Input: $N, M_0, M$                                                  Input: $N, M_0, M$
 1: $table \leftarrow$ EmptyTableOfNodes
 2: $\mathcal{T} \leftarrow \mathsf{net2transducer}(N)$
 3: $q_0 \leftarrow table.\mathsf{encode}(M_0)$
 4: $q \leftarrow table.\mathsf{encodeUpwardClosed}(M)$            $\mathcal{M} \leftarrow \{M' : M' \geq M\}$
 5: $q_{\text{old}} \leftarrow q_\emptyset$                          $\mathcal{M}_{\text{old}} \leftarrow \emptyset$
 6: while true do                                                   while true do
 7:     $q_{\text{old}} \leftarrow q$                                    $\mathcal{M}_{\text{old}} \leftarrow \mathcal{M}$
 8:     $q_{\text{pre}} \leftarrow table.\mathsf{pre}(\{q_{0\mathcal{T}}, q\}, \mathcal{T})$      $\mathcal{M}_{\text{pre}} \leftarrow pred_N(\mathcal{M})$
 9:     $q \leftarrow table.\mathsf{union}(q, q_{\text{pre}})$           $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_{\text{pre}}$
10:     if $table.\mathsf{intersection}(q_0, q) \neq q_\emptyset$ then      if $M_0 \in \mathcal{M}$ then
11:         return true                                                     return true
12:     if $q = q_{\text{old}}$ then                                     if $\mathcal{M} = \mathcal{M}_{\text{old}}$ then
13:         return false                                                    return false
```

pred is replaced by pre, and $\cup$ by union.

A slight difference between the algorithms is present in line 10. In the previous algorithm on the right we calculate if the start marking exists in the set $\mathcal{M}$. However, in the adapted algorithm we calculate the intersection of two nodes instead of simply checking if node q accepts the word q_0. The reason behind this is that some of the later benchmark instances use a set of start markings instead of the single M_0. By using intersection we can also deal with a q_0 encoding a set of markings.

The only questions remaining are about the specific mapping between nodes and markings, and the connection between the table operation pre described in Chapter 4 and the *pred* set from Algorithm 8. The next two subsections describe the encoding for the markings and the corresponding construction of the transducer for pre, before the final version of the Backwards Reachability Algorithm with the Table of Nodes is presented.

5.2.1 A Weakly Acyclic Encoding for Markings

Recall that a marking M is defined as a mapping from places of a Petri net to a natural number, which shows how many tokens are assigned to a place by the marking. We define a fixed order on the places $P = (p_1, p_2, \dots, p_n)$ of the Petri net N. Then a marking can be represented by an n-dimensional tuple $M = (m_1, m_2, \dots, m_n)$ where m_i is the number of tokens in place p_i in that marking. We define the following encoding for the markings.

Definition 5. *For a marking $M = (m_1, m_2, \dots, m_n)$ we define its* (ordinary) encoding $f(M)$ *as the language $L(\mathbf{1}^{m_1}\square\mathbf{1}^{m_2}\square\dots\mathbf{1}^{m_n}\square)$ over $\Sigma = \{\mathbf{1}, \square\}$.*

As Definition 5 illustrates, the token numbers in M are encoded unary with $\mathbf{1}$s and the different positions in M are separated with a separator symbol $\Box$. For instance, the marking $M = (2, 3, 1)$ for a Petri net with 3 places is encoded into the language $L(\mathbf{11\Box111\Box1\Box})$. This encoding maps every single marking to a single word over $\Sigma = \{\mathbf{1}, \Box\}$.

With this straight-forward representation we can construct new table operations encode and encodeUpwardClosed, which have been applied in Algorithm 9.

The encode function is sketched in Algorithm 10 and returns the node identifier of the language for the input marking M. In the adapted Backwards Reachability Algorithm it computes the node for the start marking M_0 in line 3. The function builds a string *str* into a regular expression for the language of the marking based on our encoding f, and then passes the string to create. The table operation create, which has been presented in Subsection 4.3.2, creates the node and all its successors into the table before returning its identifier.

Algorithm 10 encode

```
Input: $M = (m_1, m_2, \dots, m_n)$
1: $str \leftarrow$ empty string
2: for each $m_i \in M$ do
3:     for each $k \in \{1, \dots, m_i\}$ do
4:         $str \leftarrow str$.append(1)
5:     $str \leftarrow str$.append($\Box$)
6: return create($str$)
```

Algorithm 11 encodeUpwardClosed

```
Input: $M = (m_1, m_2, \dots, m_n)$
1: $str \leftarrow$ empty string
2: for each $m_i \in M$ do
3:     for each $k \in \{1, \dots, m_i\}$ do
4:         $str \leftarrow str$.append(1)
5:     $str \leftarrow str$.append(1*)
6:     $str \leftarrow str$.append($\Box$)
7: return create($str$)
```

There are two nested loops in Algorithm 10, with the outer one iterating over all elements of M. Inside the inner loop, we iteratively append 1 to the string *str* until the amount m_i. After all the 1s have been added for the current place p_i of the net, we insert the separator $\Box$ in line 5. Finally, we pass the complete string to the create method.

The encodeUpwardClosed operation also takes a marking M as input. Instead of creating the language of one marking, encodeUpwardClosed creates the language for the upward closed set with M as the only minimal marking. In the Backwards Reachability Algorithm it is used to create the upward closed set for the target marking in line 4 of Algorithm 9.

For the marking $M = (m_1, m_2, \dots, m_n)$, encodeUpwardClosed returns the language for the set $\{M' : m'_i \geq m_i \ \forall i \in \{1, \dots, n\}\}$. Based on the encoding in Definition 5, this is described by the language $L(\mathbf{1}^{m_1}\mathbf{1}^*\Box\mathbf{1}^{m_2}\mathbf{1}^*\Box \dots \mathbf{1}^{m_n}\mathbf{1}^*\Box)$. By adding $\mathbf{1}^*$ after every $\mathbf{1}^{m_i}$, we allow for each place p_i to have at least m_i tokens. Note here that every upward closed set is encoded into a weakly acyclic language, as the automata for expressions of this form only contain cycles which are self-loops.

Using this, a slight modification of Algorithm 10 leads to Algorithm 11 for the table operation encodeUpwardClosed. We only need to append the string 1* after the inner for loop in line 5 of Algorithm 11 and the rest of the operation stays unchanged.

5.2.2 Representing a Petri Net with a Transducer

After having fixed the encoding scheme in Definition 5 last section, we now want to develop a consistent construction of the transducer $\mathcal{T}$ which corresponds to the operation net2transducer in line 2 of Algorithm 9. The transducer must be designed in a way such that the pre of the table yields the same set of predecessor markings like $pred_N(\mathcal{M})$.

Recall the Pre language and the Predecessor set we have defined. Here, the input of the Pre language is directly expressed as a language instead of an automaton.

$$\mathrm{Pre}_{\mathcal{T}}(L) = \{w \in \Sigma^* : \exists v \in L \ s.t. \ (w, v) \in L(\mathcal{T})\}$$

$$pred_N(\mathcal{M}) = \bigcup_{t \in Tr} \{M : \exists M' \in \mathcal{M} \ s.t. \ M \xrightarrow{t} M'\}$$

We aim to design $\mathcal{T}$ in a way such that the words w in $\mathrm{Pre}_{\mathcal{T}}(L)$ are encoding the predecessor markings M in $pred_N(\mathcal{M})$:

$$f(pred_N(\mathcal{M})) = \mathrm{Pre}_{\mathcal{T}}(f(\mathcal{M}))$$

For this, the following equivalence must hold:

$$(w, v) \in L(\mathcal{T}) \iff \exists M, M' \text{ and } t \text{ such that } w = f(M), v = f(M') \text{ and } M \xrightarrow{t} M'$$

Therefore, the transducer needs to encode all transitions of the Petri net and maps the encoding of one marking M with all possible successor markings M' by firing one transition in the Petri net.

Introducing Another Encoding

A Petri net has a set of transitions Tr which are connected to the places P by the flow relation F. With the fixed order of the places $P = (p_1, p_2, \ldots, p_n)$, we can describe the effect in the places induced by firing a single transition t with a pair of n-dimensional tuples $t = (g, h)$, where g is the preset and h is the postset of transition t. In g, the element on position i, denoted by g_i, indicates how many tokens from place p_i are necessary to enable t. By firing t, this amount of tokens are removed from p_i. In h, the i-th element h_i expresses how many tokens are added into place p_i after firing t. Therefore, the complete Petri net N can be described by a set of transition pairs $t = (g, h)$.

Imagine a Petri net with a single transition $t = (g, h)$ with $g = (1, 1, 0)$ and $h = (1, 1, 3)$. Consider the marking $M = (1, 3, 2)$ for this Petri net depicted in Figure 5.2. The encoding $f(M)$ for this marking is the language of the word $w = \mathbf{1}\square\mathbf{111}\square\mathbf{11}\square$. With the marking shown in Figure 5.2, transition t_1 is enabled and the firing of t_1 leads to the marking $M' = (1, 3, 5)$ in Figure 5.3. This marking is encoded by the word $v = \mathbf{1}\square\mathbf{111}\square\mathbf{11111}\square$.

For this Petri net, the corresponding transducer needs to accept the word $(\mathbf{1}\square\mathbf{11}\square\mathbf{111}\square, \mathbf{1}\square\mathbf{11111}\square\mathbf{111}\square)$. However, the length of these words w and v is not the same but any transducer can only accept a pair of words with equal length.

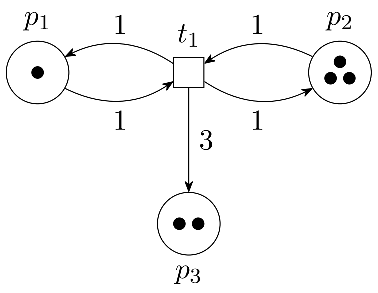


Figure 5.2: Petri Net before Firing t_1

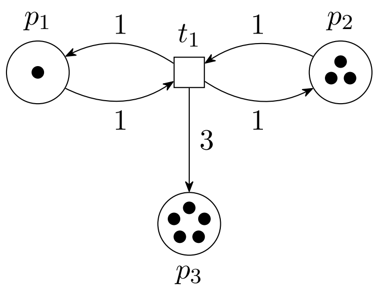


Figure 5.3: Petri Net after Firing t_1

To address this problem, we introduce a padding symbol $\Diamond$ to the alphabet with the purpose to pad w and v to the same length, and adapt the definition of the encoding.

Definition 6. *For a marking $M = (m_1, m_2, \ldots, m_n)$ we define its* padded encoding $f'(M)$ *as the language $L(\mathbf{1}^{m_1}\Diamond^*\Box\mathbf{1}^{m_2}\Diamond^*\Box\ldots\mathbf{1}^{m_n}\Diamond^*\Box)$ over $\Sigma = \{\mathbf{1}, \Box, \Diamond\}$.*

The idea is to use the ordinary encoding f for all operations in the Backwards Reachability Algorithm except of the ones involving the transducer $\mathcal{T}$. We will define $\mathcal{T}$ for a transition t so that it will map specific padded encodings w to other specific padded encodings v. Only for the construction of the transducer and the pre procedure, we represent a node with these specific padded encodings.

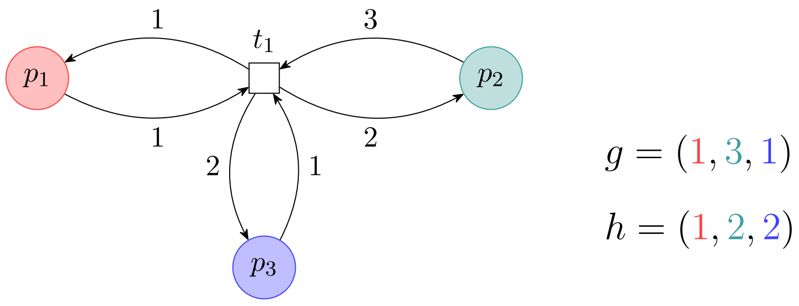
Construction of the Transducer

With the introduced padded encoding from Definition 6 we are constructing a transducer $\mathcal{T}$ for a Petri net transition t. The following property should be satisfied:

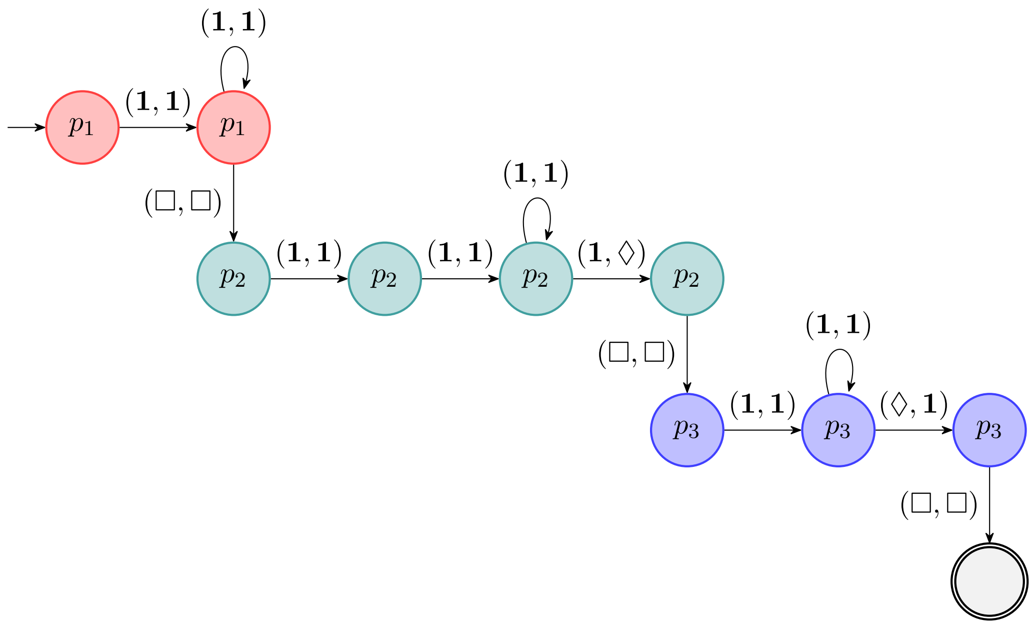
- For all markings M, M' and transition t, where $M \xrightarrow{t} M'$, there exist some padded encodings $w \in f'(M)$ and $v \in f'(M')$ such that $\mathcal{T}$ accepts (w, v).

Consider a Petri net with one transition $t_1 = ((1,3,1),(1,2,2))$ depicted in Figure 5.4a. For this Petri net the constructed transducer is illustrated in Figure 5.4b.

We can have a closer look at the structure of the transducer. In Figure 5.4a, we have three places, which correspond to the three parts of the transducer with different levels and colors. Each level reads in an amount of tokens for the respective place. The different levels are connected by reading in a pair of separators $(\Box, \Box)$, which are the vertical transitions in Figure 5.4b. The final $(\Box, \Box)$ from the last level leads into the sole accepting state of the transducer.



(a) Petri Net with One Transition t_1



(b) Transducer $\mathcal{T}$ for Transition t_1

Figure 5.4: From Petri Net to Transducer

Given the transition $t = (g, h)$, we can distinguish between two cases for constructing the part of the transducer for the place p_i.

Case 1: $g_i \geq h_i$**.** Here the number g_i of tokens removed from the place is higher or equal compared to the number h_i of tokens added by firing t. In the example in Figure 5.4 this can be seen for place p_2, where $g_2 = 3$ and $h_2 = 2$.

The expression the transducer accepts for this case is of the form $(\mathbf{1},\mathbf{1})^{h_i}(\mathbf{1},\mathbf{1})^*(\mathbf{1},\Diamond)^{g_i-h_i}$. This expression can be decomposed into three subexpressions.

The first subexpression is of the form $(\mathbf{1},\mathbf{1})^{h_i}$. Before the execution of t, there need to be at least g_i tokens inside p_i, and after the execution, there are still at least h_i tokens in the place. This means both before and after firing t, the place contains a minimum of h_i

tokens. Therefore, the transducer first reads in h_i many $(\mathbf{1},\mathbf{1})$ pairs. In Figure 5.4b, this corresponds to the 2 transitions ($h_2 = 2$) with $(\mathbf{1},\mathbf{1})$ in the beginning of the middle level of the transducer.

The second part of the expression is $(\mathbf{1},\mathbf{1})^*$, which corresponds to the self-loop of the middle level in Figure 5.4b. This allows the place to have an arbitrary amount of further tokens.

The final subexpression $(\mathbf{1},\Diamond)^{g_i-h_i}$ adds padding symbols. After firing the transition, the number of tokens in place p_i decreases by $g_i - h_i$. Thus, we pad $g_i - h_i$ many $\mathbf{1}$s in the first position (marking before firing t) by as many $\Diamond$s in the second position (marking after firing t). This is shown by the one transition ($g_2 - h_2 = 1$) with $(\mathbf{1},\Diamond)$ at the end of the middle level for p_2 in Figure 5.4b.

Case 2: $g_i \leq h_i$**.** This case is symmetrical to the previous one. The only difference is in the third fragment of the transducer. Instead of $(\mathbf{1},\Diamond)$ we read in $(\Diamond,\mathbf{1})$, because after firing the transition the number of tokens in p_i increases. Therefore, the expression recognized for this case is of the form $(\mathbf{1},\mathbf{1})^{g_i}(\mathbf{1},\mathbf{1})^*(\Diamond,\mathbf{1})^{h_i-g_i}$. In the concrete example in Figure 5.4, this case can be seen in the bottom level for p_3.

preprocess and postprocess

As mentioned, the constructed transducer only recognizes specific padded encodings dependent on the transition. In the concrete example in Figure 5.4b the transducer only maps a word of the form $w = \mathbf{1}\mathbf{1}^k\Box\mathbf{1}\mathbf{1}\mathbf{1}^k\mathbf{1}\Box\mathbf{1}\mathbf{1}^k\Diamond\Box$ to another word $v = \mathbf{1}\mathbf{1}^k\Box\mathbf{1}\mathbf{1}\mathbf{1}^k\Diamond\Box\mathbf{1}\mathbf{1}^k\mathbf{1}\Box$. However, besides the construction of the transducer in line 2 of Algorithm 9, we only use the padded encodings for the pre operation in line 8. The rest of the algorithm is operating with the ordinary encoding from Definition 5.

Therefore, further table operations preprocess and postprocess are required to convert between nodes of the specific padded encoding and nodes of the ordinary encoding. Since the transducer is only used in the pre function, these two new operations are only performed before and after pre.

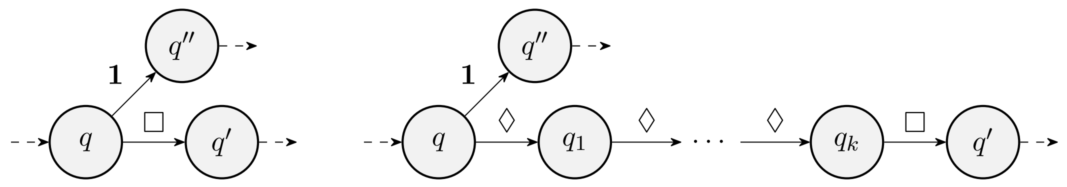
preprocess converts a node in the ordinary encoding into an equivalent node in the padded encoding specific to the image of the transducer by adding $\Diamond$ transitions. The node returned by preprocess is then passed on to the pre operation. postprocess converts the result of pre, which is a node in the specific padded encoding of the preimage of the transducer, back to the correct ordinary encoding, removing $\Diamond$ transitions in the process.

The question remains, how many padding transitions need to be added or removed. As can be seen in Figure 5.4, we construct the transducer in a way such that the amount of paddings before the i-th separator $\Box$ is fixed for both preimage and image of the transducer. Therefore, we can use this property to add or remove that fixed amount of $\Diamond$ symbols. Taking the transducer in Figure 5.4b, preprocess needs to add one $\Diamond$ before the second $\Box$, while postprocess will remove one $\Diamond$ before the third $\Box$ for this concrete case.

Figure 5.5 visualizes the preprocess operation. Figure 5.5a on the left shows a subautomaton preprocess is currently processing starting with the node q. A second input given to preprocess is a counter i, which indicates that the next $\Box$ is the i-th separator symbol encountered so far. There are only successors for $\mathbf{1}$ and $\Box$, because the automaton given to preprocess is still in the ordinary encoding and does not contain any $\Diamond$s.

Assume that in the image of the transducer there are k padding symbols before the i-th separator symbol. Then preprocess needs to add k new nodes $q_1, \ldots, q_k$ between q and q', as depicted on the right in Figure 5.5b. The $\Box$ transition of q is removed and one $\Diamond$ transition from q to q_1 is added. From q_1 to q_k there are $k-1$ further $\Diamond$ transitions. Eventually, the $\Box$ transition is inserted from q_k to q'. Finally, preprocess makes recursive calls for q'' and q'. Note here that for the call with q'' we will pass on the same counter i, but for the call with q' we increase the counter by one, as the next $\Box$ after q' will be the $i+1$-th separator.

The postprocess operation works in the reverse direction from Figure 5.5b to Figure 5.5a. Instead of adding new nodes, we remove $q_1, \ldots, q_k$ by removing the $\Diamond$ transition from q and adding a $\Box$ transition from q directly to q'.



(a) before preprocess (b) after preprocess

Figure 5.5: Visualization of preprocess

Final Backwards Reachability Algorithm with the Table of Nodes

With the described construction of the transducer and the two new table operations preprocess and postprocess, we achieve a final version of the Backwards Reachability Algorithm shown in Algorithm 12. The difference to the previous version in Algorithm 9 is again highlighted in blue.

In line 2, instead of constructing one transducer we build a list $\mathcal{T}^*$ of transducers, where each $\mathcal{T} \in \mathcal{T}^*$ represents one transition of the Petri net N. Therefore, we need to add an inner loop in line 8 which iterates over all $\mathcal{T}$. Inside this loop, we use preprocess and postprocess before and after the pre operation. In line 9, preprocess converts the ordinary encoded q into the node q' in the padded encoding. After the pre operation returns q_{pre}, which is also in the form of a padded encoding, we use postprocess to convert back into the ordinary encoding returning node q''.

Algorithm 12 Final Version of Backwards Reachability Algorithm with Table of Nodes

```
Input: $N, M_0, M$
 1: $table \leftarrow$ EmptyTableOfNodes
 2: $\mathcal{T}^* \leftarrow \mathsf{net2transducers}(N)$
 3: $q_0 \leftarrow table.\mathsf{encode}(M_0)$
 4: $q \leftarrow table.\mathsf{encodeUpwardClosed}(M)$
 5: $q_{\text{old}} \leftarrow q_\emptyset$
 6: while true do
 7:     $q_{\text{old}} \leftarrow q$
 8:     for each $\mathcal{T} \in \mathcal{T}^*$ do
 9:         $q' \leftarrow table.\mathsf{preprocess}(q, \mathcal{T})$
10:         $q_{\text{pre}} \leftarrow table.\mathsf{pre}(\{q_{0\mathcal{T}}, q'\}, \mathcal{T})$
11:         $q'' \leftarrow table.\mathsf{postprocess}(q_{\text{pre}}, \mathcal{T})$
12:         $q \leftarrow table.\mathsf{union}(q, q'')$
13:     if $table.\mathsf{intersection}(q_0, q) \neq q_\emptyset$ then
14:         return true
15:     if $q = q_{\text{old}}$ then
16:         return false
```

6 Evaluation

We implemented the Backwards Reachability Algorithm with the Table of Nodes, sketched in Algorithm 12, in a tool PETRIMAT using the programming language C++.

The tool expects the Petri net N with start and target marking (M_0, M) in the MIST format as input [11]. We implemented a parser to transform from the MIST format into the representation we use in our implementation.

There are some differences in the instances of the MIST format compared to the input expected by Algorithm 12. First, the problem instances sometimes contain a set of start markings instead of the single marking M_0. To handle this scenario, we adapt the `encode` operation to generate the correct node given a set of markings.

Furthermore, the MIST format allows to have multiple target markings M to be covered. The instance is coverable, if one of the target markings is coverable. We simply encode each single target marking analogous to line 4 in Algorithm 12 and finally take the union of all these nodes for obtaining $\mathcal{M}$.

We compare PETRIMAT (*ptm*) with the existing tools MIST (*mist*) [11] and QCOVER (*qcov*) [2]. The tools were tested on existing coverability instances with a benchmark script from the tool QCOVER [2]. All benchmarks were run on a machine with an Apple M1 Pro CPU and 32 GiB RAM with a timeout of 300 seconds (5 minutes). The set of coverability problems used are from the toolkit of MIST and consist of 27 coverability instances.

The benchmark results for the instances are shown in Table 6.1. The instances are ordered ascendingly by the number of places in the Petri net. The first column shows the name of the coverability instance, while the second column depicts the result of the coverability problem. Here, safe means that the given instance is not coverable, and unsafe implies that the instance is coverable. Out of the 27 instances, only 4 are unsafe and the rest are all safe.

The next two columns show the amount of places and transitions in the underlying Petri net. Finally, the solve time and an estimation for the maximal memory usage of the three tools are shown. Both the solve time and the memory usage are computed using the built-in `time` command of the Z shell (zhs). For the solve time, the user time and system time are added together to get the CPU time the tools consume.

First, we can observe there is no consistent relation between the number of places and transitions with the difficulty of the instance. For example, the `mesh3x2` instance with over 50 places and transitions is solved by all three tools in under 4 seconds, while the `kanban` instance with 16 places and transitions takes around 159 seconds for *mist* and even times out for *qcov*.

		Petri net		**solve time (ms)**			**max memory (MiB)**		
instance	**result**	places	trans	*ptm*	*mist*	*qcov*	*ptm*	*mist*	*qcov*
`basicME`	safe	5	4	10	8	3039	1.53	4.53	78.02
`pingpong`	safe	6	6	13	8	2383	1.55	4.59	75.31
`newrtp`	safe	9	12	33	10	2054	1.73	4.55	70.63
`lamport`	safe	11	9	51	13	2673	2.41	4.59	80.28
`MultiME`	safe	12	11	57	11	2797	2.39	5.05	84.52
`read-write`	safe	13	9	288	56	2300	4.3	5.95	72.56
`manufacturing`	safe	13	6	2454	885	2541	26.31	11.75	77.5
`csm`	safe	14	13	93	22	2379	3.06	5.29	72.92
`peterson`	safe	14	12	144	21	3217	3.61	4.79	76.38
`leabasicapproach`	unsafe	16	12	40	11	3229	2.83	4.8	78.7
`newdekker`	safe	16	14	1167	56	2620	16.13	4.71	78.02
`kanban`	unsafe	16	16	7503	158863	-	139.77	52.45	90.5
`kanban_bounded`	safe	16	16	8774	160560	2560	135.05	4.6	73.41
`multipool`	safe	18	21	221	286	2640	4.61	7.6	70.73
`fms`	safe	22	20	799	36	2569	6.23	4.66	75.25
`fms_attic`	safe	22	20	16381	827	2646	57.86	8.44	76
`extendedread-write-sc`	safe	24	22	15508	94171	3383	80.69	5.46	74.69
`extendedread-write`	safe	24	22	-	-	2900	4089.64	33.29	83.95
`bingham_h25`	safe	28	51	1525	248	2561	14.86	7.55	73.16
`pncsasemiliv`	unsafe	31	36	6133	836	3810	89.11	6.37	79.22
`pncsacover`	unsafe	31	36	128721	-	31920	1493.59	15.17	85.36
`mesh2x2`	safe	32	32	512	129	2960	8.7	7.06	74.67
`mesh3x2`	safe	52	54	4000	1381	2677	51.52	11.75	70.91
`bingham_h50`	safe	53	101	11059	3461	3047	92.69	14.46	71.22
`bingham_h150`	safe	153	301	-	-	3336	1984.59	102.02	76.78
`bingham_h250_attic`	safe	253	501	127682	9902	93454	733.72	454.57	231.86
`bingham_h250`	safe	253	501	-	-	3127	3257.52	106.34	82.75

Table 6.1: Results of the Benchmarks

Looking at the time columns, we can see that *ptm* is able to solve 24 out of the 27 instances within the timeout of 300 seconds, while *mist* finds the solution for 23 and *qcov* for 26 instances.

Comparing the times more closely, we recognize that there is similar trend of the solve times from *ptm* and *mist*. Still, there exist some problems where *ptm* performs significantly better, like `kanban` or `extendedread-write-sc`, and others where *mist* solves the problems a lot faster, which is the case for `newdekker` or `fms_attic`. But the overall fluctuations are mostly consistent between these two tools. *qcov* on the other hand shows very stable times for most of the instances, being mostly between 2 and 4 seconds. Only for `pncsacover`, `bingham_h250_attic` and `kanban` *qcov* needs more than 4 seconds. The reason for this pattern could be that *qcov* needs a certain amount of processing time independent of the instance structure. In general, *ptm* and *mist* are faster in a lot of smaller problems but have more difficulties processing some more demanding instances, where they are significantly slower than *qcov*.

Looking at the memory usage for the three tools, we can observe a clear positive correlation to the solve times. The timed out instances of *ptm* consume gibibytes of memory. Similar to the solve times, the memory consumption for *qcov* stays very stable around 80 mebibytes. The memory measured of *mist* is very low with only a few mebibytes for most of the instances and even the highest value measured is below 500 mebibytes.

Altogether, the solve time of our tool *ptm* is competitive to the other two tools. While in the smaller instances, *ptm* is often a little slower than *mist*, it solves the more difficult `kanban` and `kanban_bounded` instances roughly 20 times faster than *mist*. Against *qcov*, our *ptm* has a clear advantage for solving smaller instances and is also much faster for the `kanban` instance. The overall memory measurements show good results for *ptm* in the easier instances, but they are an enormous amount higher than the other two tools in numerous instances like `extended-read-write`, `bingham_h150`, or `pncsacover`.

However, it is questionable whether the memory command of the Z shell gives an accurate representation of the peak memory used by the tools. For instance, the Table of Nodes used in *ptm* does not have any operations for removing nodes. During the procedure a lot of nodes are never used again but still remain in the table and are thus contributing to the memory consumption.

To have a better comparison of the memory usage we want to take a closer look at the size of the underlying data structures. For our own tool *ptm* we can collect information about the Table of Nodes very easily. The *mist* tool uses a data structure called Interval Sharing Tree to represent the minimal markings of the predecessor set [12]. Running the tool with a special command allows insights on the size of the generated tree at the end. The tool *qcov* is left out in these comparisons, as we have not found a good way to compare its memory consumption fairly with the other two tools.

Table 6.2 depicts more information about the data structures from *ptm* and *qcov*. The four columns under *ptm* in Table 6.2 show properties of our implementation with the Table of Nodes. The column iter shows the number of iterations before the Backwards Reachability Algorithm terminates, which for these instances stays mostly below 30.

The table size shows the complete amount of nodes in the table, which have been created during the execution of Algorithm 12. We can see that the numbers range from hundreds to millions of generated nodes. 9 out of the 27 instances generate more than 200000 nodes into the table.

The column max node in the middle indicates the size of the largest node q after any iteration of the Backwards Reachability Algorithm from Algorithm 12. The size we mean in this context is the amount of states of the automaton for q. This value is a better estimation for the peak memory used during the computation than the complete table size, as this represents the maximal automaton size the algorithm is processing at any point in the

	solve time (ms)		***ptm***				***mist***
instance	*ptm*	*mist*	iter	table size	max node	end node	end size
`basicME`	10	8	3	365	31	31	26
`pingpong`	13	8	4	427	28	28	33
`newrtp`	33	10	6	939	31	29	50
`lamport`	51	13	5	3210	97	62	80
`MultiME`	57	11	5	3113	73	66	70
`manufacturing`	288	56	14	89483	3348	1489	953
`read-write`	2454	885	9	9923	322	236	235
`peterson`	93	22	6	6799	210	181	169
`csm`	144	21	5	5210	110	93	109
`leabasicapproach`	40	11	2	4030	180	180	162
`newdekker`	1167	56	8	51804	1468	895	341
`kanban`	7503	158863	11	571617	6635	1132	1594
`kanban_bounded`	8774	160560	18	574675	6635	768	720
`multipool`	221	286	6	11731	137	116	196
`fms`	799	36	15	15827	285	247	233
`fms_attic`	16381	827	27	228830	2735	2431	881
`extendedread-write-sc`	15508	94171	24	322684	2769	1245	758
`extendedread-write`	-	-	-	-	-	-	-
`bingham_h25`	1525	248	27	48969	161	161	253
`pncsasemiliv`	6133	836	7	340348	6375	1652	2139
`pncsacover`	128721	-	15	7015463	80180	779	-
`mesh2x2`	512	129	5	30142	271	271	344
`mesh3x2`	4000	1381	8	202945	914	796	987
`bingham_h50`	11059	3461	52	316044	311	311	503
`bingham_h150`	-	-	-	-	-	-	-
`bingham_h250_attic`	127682	9902	4	2869137	3547	2914	4512
`bingham_h250`	-	-	-	-	-	-	-

Table 6.2: Information on the Data Structure

algorithm. Comparing this column with the table size, we can see for most instances there is a difference of a factor over 30.

The end node is the size of the node q from Algorithm 12 right before terminating. For the uncoverable safe instances, this is the size needed to represent the fully computed predecessor set containing all backwards reachable markings. We can observe that for most instances, the size is in a similar magnitude as the max node size. There are some cases, like `pncsacover` or `kanban_bounded`, where the end size is much smaller compared to the max node size.

Finally, the last column of Table 6.2 beneath *mist* depicts the size of the final Interval Sharing Tree of *mist* also representing the set of predecessors. We can compare the end node size of *ptm* with the end size of *mist* to get an overview, how much memory the respective data structures need to represent the same predecessor set of markings. It is noticeable that for all instances the respective sizes are in a very similar order of magnitude. There are some cases, where the tree of *mist* has a smaller representation, like for `newdekker` or `fms_attic`, which are also the instances *mist* is much faster than *ptm*. But for `bingham_250_attic` or `mesh3x2` the underlying automaton of *ptm* is more compact.

We performed further tests where we record the size of the table and the size of the node q in Algorithm 12 after each iteration of the while loop. Figure 6.1 and Figure 6.2 show the development of these sizes for some instances.

Figure 6.1 shows how the size of the node q, characterizing the predecessors, evolves from iteration to iteration. We can observe that almost all of the depicted instances have a steep rise in the size during the first couple of iterations. After the maximal peak size is reached, which is exactly the max node in Table 6.2, the size gradually begins to fall again. The speed of the decline is varying for the instances, with the size of `pncsacover` or `kanban_bounded` falling by a greater amount and speed compared to `extended-read-write-sc`. Only the plot of the `bingham_25` instance behaves very differently to the others. The size of q slowly increases here with more iterations and in the end there is even a small leap before the algorithm terminates. Overall, the trend of the curves matches our intuition that intermediate states of the algorithm have more complicated representations, while the end representation

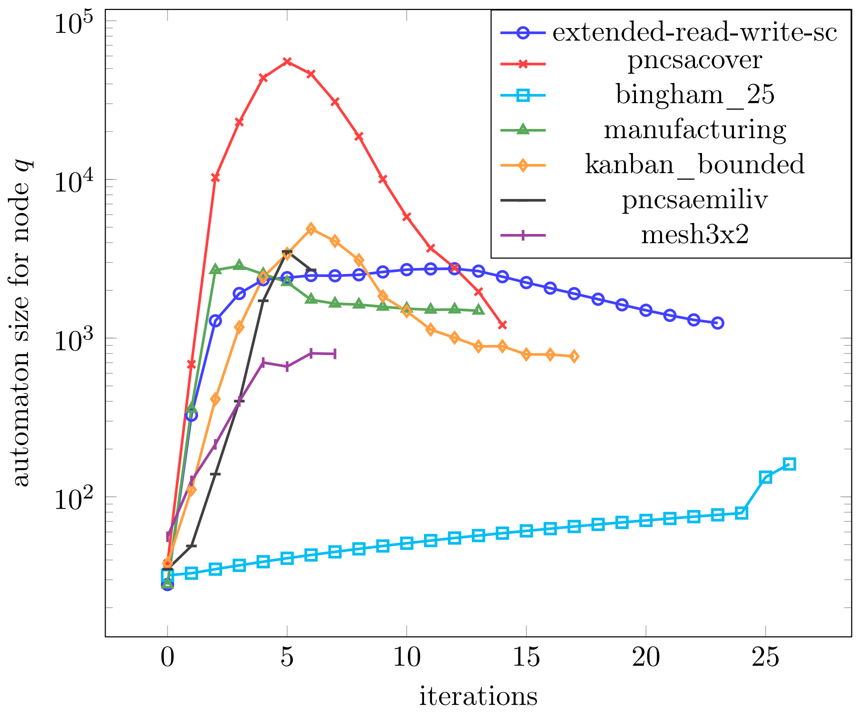


Figure 6.1: Growth of the Size of q

is more simple.

Figure 6.2 is sketching the growth of the table with the iterations. For all instances we can perceive logarithmic growth curves. A maximum point is reached in the first couple of iterations, and after that the table size is slowly approaching a limit. Comparing this with the growth of q, we can see that the peaks in Figure 6.1 correspond with the iterations where the steep growth of the curves in Figure 6.2 is subsiding.

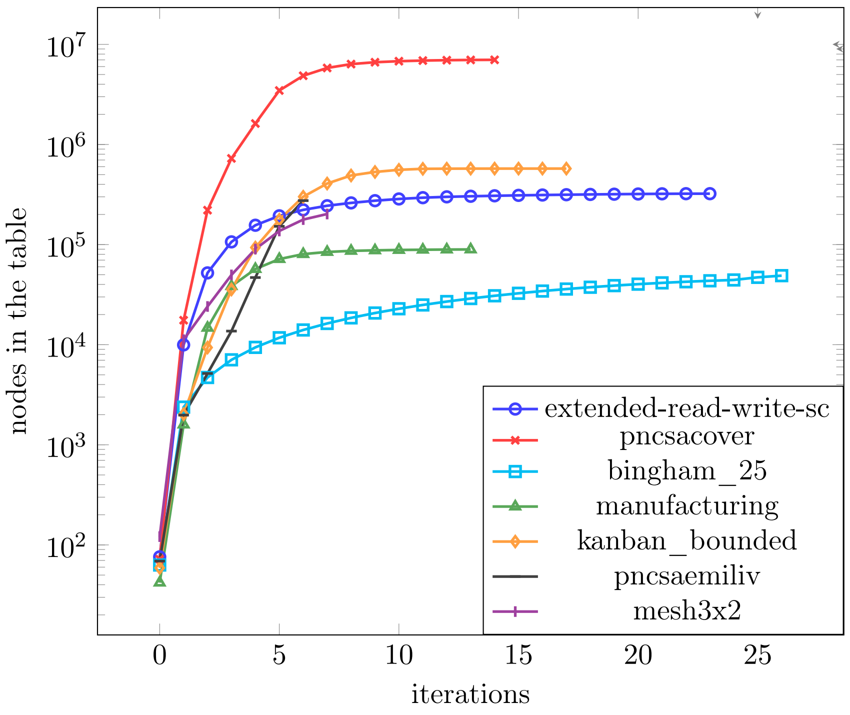


Figure 6.2: Growth of the Size of Table

Computing Minimal Markings for q

We have already shown formally in Section 4.5 that our pre operation computes the Pre language, and with Chapter 5 we have illustrated how we implement the Backwards Reachability Algorithm using the concepts from Chapter 4. However, it is very desirable to validate the formal correctness of our tool by observing unanimous results with other coverability tools for complex problem instances.

For all the benchmarks which we performed and did not time out, the coverability decision results of our tool always aligned with the other two tools *mist* and *qcov*, which is already a reassuring outcome.

To verify our tool on a more thorough level, we have implemented another operation for the table called sanity. This operation takes a node q in the table and computes the set of minimal markings for the upward closed set q represents. In the following we provide a simple sketch on how the procedure works.

The operation starts by computing a topological order on the nodes for the automaton of q corresponding to the relation $\preceq$ from Section 4.1. After this, we iterate over each node starting from q by this topological order. Since we have knowledge about the encoding (Definition 5), we can easily map back from each node to the part of the marking it is characterizing by remembering the amount of $\mathbf{1}$ and $\square$ transitions we have encountered so far. Here, we can ignore all the self-loop transitions, because they do not play a relevant role for the minimal markings. With this approach, at each node we collect a set of markings, extract the minimal markings of this set by removing non-minimal elements, and pass the minimized set over to the successors of the node. When we reach the accepting node q_ϵ at last, the set of markings here are the minimal markings represented by q.

By passing a particular command to Mist, the tool can also print the minimal markings it has computed for an instance. We can compare the equality of the minimal markings returned by sanity and the ones printed out by Mist to verify if they have computed the same set of predecessors for the instances. For some smaller instances, we have witnessed equal sets with side-by-side comparing and for more larger ones we compared the equality of the number of minimal markings, which were also the same. Only for the unsafe instances we observed differing values, as the tools might terminate earlier and are not computing the complete predecessor set.

7 Conclusion

In this thesis, we have developed a new approach for solving the coverability problem in Petri nets, which incorporates the class of weakly acyclic languages. With the idea of using weakly acyclic automata for representing markings in Petri nets, we have implemented the abstract procedure of the Backwards Reachability Algorithm using these automata.

For this, we designed a data structure, which we call Table of Nodes, to store weakly acyclic languages efficiently. Furthermore, we implemented operations for this data structure, which are required for the Backwards Reachability procedure. For the main operation pre, responsible for calculating the language of the predecessor markings in each iteration of the algorithm, we give a short proof that it computes the correct result.

Hereafter, we developed an encoding scheme to map from a Petri net marking to the corresponding weakly acyclic language. We capture the structure of the Petri net by a transducer, which is given as input for the pre operation.

Using these constructs, we were able to successfully implement the Backwards Reachability Algorithm with the Table of Nodes into the tool Petrimat.

Finally, we performed benchmarks with existing coverability instances and compared our implementation to two other state-of-the-art coverability tools. The results indicate that our implementation is competitive with those tools and even outperforms them for some instances. Further tests giving information on our data structure show that our approach with the use of automata offers a compact representation for the final set of predecessors.

We checked the correctness of our tool not only by comparing the coverability results to the other tools, but we have also implemented an operation which for a given node computes the minimal markings this node represents. By comparing the minimal markings obtained by our tool with the minimal markings obtained by one other tool, we could verify that the computed predecessor set is the same for all uncoverable instances.

At last, we conclude by giving some future directions which can be further examined. In order to have a better overview where our implementation is spending a lot of time, an analysis with the use of a profiler should be considered. With these insights, more targeted optimizations can be performed that might increase the speed of the current implementation by a great amount.

Furthermore, the Table of Nodes lacks an operation for removing nodes, which burdens the memory usage of our tool. To address this issue, a garbage collection procedure can be implemented for discovering and freeing the memory of unused nodes.

List of Figures

List of Algorithms

List of Tables

Bibliography

[1] P. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. “General decidability theorems for infinite-state systems.” In: *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 1996, pp. 313–321. DOI: `10.1109/LICS.1996.561359`.

[2] M. Blondin. *QCover*. `https://github.com/blondimi/qcover`. 2021.

[3] M. Blondin, M. Cadilhac, and J. Esparza. “Symbolic representation of weakly acyclic sets.” unpublished. N.D.

[4] R. V. Carvalho, F. J. Verbeek, and C. J. Coelho. “Bio-modeling Using Petri Nets: A Computational Approach.” In: *Theoretical and Applied Aspects of Systems Biology*. Ed. by F. Alves Barbosa da Silva, N. Carels, and F. Paes Silva Junior. Cham: Springer International Publishing, 2018, pp. 3–26. ISBN: 978-3-319-74974-7. DOI: `10.1007/978-3-319-74974-7_1`.

[5] S. Cherdal and S. Mouline. “Modelling and Simulation of Biochemical Processes Using Petri Nets.” In: *Processes* 6.8 (2018). ISSN: 2227-9717. DOI: `10.3390/pr6080097`.

[6] Y. U. D. of Computer Science and R. Lipton. *The reachability problem requires exponential space*. Research report (Yale University. Department of Computer Science). Department of Computer Science, Yale University, 1976.

[7] G. Delzanno and J.-F. Raskin. “Symbolic Representation of Upward-Closed Sets.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by S. Graf and M. Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 426–441. ISBN: 978-3-540-46419-8.

[8] G. Delzanno, J.-F. Raskin, and L. V. Begin. “Covering sharing trees: a compact data structure for parameterized verification.” In: *International Journal on Software Tools for Technology Transfer* 5 (2004), pp. 268–297.

[9] J. Esparza. *Petri Nets Lecture Notes*. 2019.

[10] A. Finkel. “The Minimal Coverability Graph for Petri Nets.” In: vol. 674. June 1991, pp. 210–243. ISBN: 978-3-540-56689-2. DOI: `10.1007/3-540-56689-9_45`.

[11] P. Ganty. *mist - a safety checker for Petri Nets and extensions*. `https://github.com/pierreganty/mist`. 2015.

[12] P. Ganty, C. Meuter, L. Begin, G. Kalyon, J.-F. Raskin, and G. Delzanno. “Symbolic Data Structure for sets of k-uples of integers.” In: (2007).

[13] S. M. German and A. P. Sistla. “Reasoning about systems with many processes.” In: *J. ACM* 39.3 (1992), 675–735. ISSN: 0004-5411. DOI: `10.1145/146637.146681`.

[14] K. M. van Hee, N. Sidorova, and J. M. van der Werf. “Business Process Modeling Using Petri Nets.” In: *Transactions on Petri Nets and Other Models of Concurrency VII*. Ed. by K. Jensen, W. M. P. van der Aalst, G. Balbo, M. Koutny, and K. Wolf. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 116–161. ISBN: 978-3-642-38143-0.

[15] A. Kaiser, D. Kroening, and T. Wahl. “A Widening Approach to Multithreaded Program Verification.” In: *ACM Trans. Program. Lang. Syst.* 36.4 (2014). ISSN: 0164-0925. DOI: `10.1145/2629608`.

[16] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. “Difference Decision Diagrams.” In: *Computer Science Logic*. Ed. by J. Flum and M. Rodriguez-Artalejo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 111–125. ISBN: 978-3-540-48168-3.

[17] C. Rackoff. “The covering and boundedness problems for vector addition systems.” In: *Theoretical Computer Science* 6.2 (1978), pp. 223–231. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(78)90036-1`.

[18] M. Silva and E. Teruel. “Petri Nets for the Design and Operation of Manufacturing Systems.” In: *European Journal of Control* 3.3 (1997), pp. 182–199. ISSN: 0947-3580. DOI: `https://doi.org/10.1016/S0947-3580(97)70077-3`.

[19] P. Wenzelburger and F. Allgöwer. “A Petri Net Modeling Framework for the Control of Flexible Manufacturing Systems ∗∗The authors wish to acknowledge the funding provided by the Federal Ministry of Education and Research Germany – Research Campus ARENA2036.” In: *IFAC-PapersOnLine* 52.13 (2019). 9th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2019, pp. 492–498. ISSN: 2405-8963. DOI: `https://doi.org/10.1016/j.ifacol.2019.11.111`.

[20] P. Wolper and B. Boigelot. “Verifying systems with infinite but regular state spaces.” In: *Computer Aided Verification*. Ed. by A. J. Hu and M. Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 88–97. ISBN: 978-3-540-69339-0.