Algoritmos para el conjunto de mínima cobertura de las Redes de Petri

Algorithms for minimal coverability set of Petri Nets

Memoria que se presenta para el Trabajo de Fin de Máster

Sara Antón Fernández

Dirigido por

David de Frutos Escrig
Fernando Rosa Velardo

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática, Universidad Complutense de Madrid
Madrid, Septiembre 2023

Abstract

Since Carl Adam Petri defined Petri Nets in the 1960s, they have been a highly useful tool for modeling and analyzing concurrent systems, serving as a graphical representation that describes the dynamics of complex systems, making them an essential asset in computer science research.

One of the central challenges in the analysis of Petri nets is determining their behavior and their ability to reach different states. The coverability tree is a crucial data structure in this context as it efficiently represents a compact approximation of the set of states a Petri net can reach during its execution. However, calculating the minimal coverability tree of a Petri net, which provides a more concise and simplified view of its behavior, is a computationally challenging task.

In this work, we will explore the algorithms of Finkel, CovProc, a minimal coverability set algorithm that uses Tarjan's algorithm (StackProc), Monotonic-Prunning and MinCov developed for the purpose of calculating the minimal coverability set of a Petri Net. Each of these approaches has been designed to address this problem from a different perspective, aiming to optimize efficiency and result accuracy. These five algorithms will be examined in detail, presenting their theoretical foundations and analyzing their effectiveness and efficiency. Thus, we will have a comprehensive view of the algorithms that have been developed to calculate the minimal coverability set of Petri nets.

Resumen

Desde que Carl Adam Petri definió las Redes de Petri en los años 60, estas han sido una herramienta muy utilizada para modelar y analizar sistemas concurrentes, sirviendo como una representación gráfica que describe la dinámica de sistemas complejos, lo que las convierte en un recurso esencial en las Ciencias de la Computación.

Uno de los problemas centrales en la análisis de redes de Petri es la determinación de su comportamiento y su capacidad de alcanzar diferentes estados. El árbol de cobertura, es una estructura de datos crucial en este contexto, ya que representa de manera eficiente una aproximación compacta útil del conjunto de estados a los que puede llegar una red de Petri durante su ejecución. Sin embargo, calcular el árbol de mínima cobertura de una red de Petri, que proporciona una visión más concisa y simplificada de su comportamiento, es una tarea computacionalmente desafiante.

En este trabajo exploraremos los algoritmos de Finkel, CovProc, un algoritmo que computa el conjunto de mínima cobertura utilizando el algoritmo de Tarjan (StackProc), Póda-Monótona y MinCov, desarrollados con el propósito de calcular el conjunto de mínima cobertura de una Red de Petri. Cada uno de estos enfoques ha sido diseñado para abordar este problema desde una perspectiva diferente, buscando optimizar la eficiencia y la precisión del resultado. Estos cuatro algoritmos se examinarán en detalle presentando sus fundamentos teóricos y analizando su eficacia y eficiencia. Así, tendremos una visión completa de los algoritmos que se han desarrollado para calcular el conjunto de mínima cobertura de las redes de Petri.

Key words

Petri Nets, coverability set, minimal coverability set, Finkel's algorithm, Cov-Proc algorithm, Tarjan's algorithm, Monotonic-Pruning algorithm, MinCov algorithm

Palabras clave

Redes de Petri, conjunto de cobertura, conjunto de mínima cobertura, algoritmo de Finkel, algoritmo de CovProc, algoritmo de Tarjan, algoritmo de Poda-Monótona, algoritmo MinCov

Índice general

Capítulo 1

Introducción

A lo largo de los últimos treinta años se han presentado diversos algoritmos para computar el conjunto de cobertura de las Redes de Petri. La importancia de este reside en que representa una aproximación compacta útil del conjunto de marcajes, cubriendo todos los marcajes alcanzables de la Red de Petri. Así, se puede saber el comportamiento de la red, cuáles de sus lugares están acotados y cuáles no, qué estados son alcanzables y cuáles son inalcanzables, etc.

En este trabajo se van a presentar y analizar los algoritmos que se han desarrollado para realizar esta labor.

Un punto de partida para la búsqueda de estos algoritmos fue el de Karp-Miller [1]. Este computa un árbol de cobertura, que no es necesariamente único. Para construir el árbol de Karp-Miller de una Red de Petri tomamos el marcaje inicial de dicha red y se lo asignamos al nodo raíz del árbol. A partir de él y disparando transiciones, vamos alcanzando nuevos marcajes y construyendo el árbol. Este se basa en la monotonía de las Redes de Petri, que es una propiedad que viene a decir que si a partir de un marcaje M_1 y una transición t se alcanza un marcaje M_1' entonces, a partir de un marcaje $M_2 \geq M_1$ entonces se puede disparar t desde M_2 llegando a un marcaje $M_2' \geq M_1'$. Con motivo de esto se llevan a cabo las aceleraciones, de manera que se "saturan" a infinito los lugares del marcaje M_2 que tienen más tokens que los del marcaje M_1, y dejando intactos los demás lugares. De esta manera se ha construido un ω-marcaje.

Finkel [1] se basó en este algoritmo para desarrollar uno que computase el árbol de mínima cobertura de una Red de Petri. A diferencia del árbol de cobertura, el de mínima cobertura es único. En su algoritmo, Finkel intro-

dujo una estrategia de poda muy agresiva lo que hizo que, aunque esto se vió mucho después con un contraejemplo, el algoritmo no fuera completo en todos los casos es decir, no computa el conjunto de mínima cobertura, sino una aproximación.

En este punto había dos opciones para abordar el problema de mínima cobertura de las Redes de Petri: intentar arreglar el algoritmo de Finkel o partir de cero. Esto último fue lo que hicieron Raskin, Geeraerts y Van Begin, presentando el algortimo CovProc [2]. Definieron la secuencia de cobertura y trabajaron con conjuntos de pares de marcajes. Este novedoso algoritmo consiguió computar de manera correcta el conjunto de mínima cobertura de una Red de Petri.

A partir de este momento se intentó encontrar nuevos algoritmos que computen el conjunto de mínima cobertura de manera más eficiente y es entonces cuando Piipponen y Valmari [3], Reynier y Servais [4], y por último, Haddad, Finkel y Khmelnitsky [5] presentan tres algoritmos.

El primero, basado en el algoritmo de Tarjan, sigue la línea de CovProc pero utilizando conjuntos de marcajes en lugar de pares. Por otro lado, con el algoritmo de Poda-Monótona se vuelve a la idea inicial de Finkel pero, a diferencia de este, su estrategia de poda es menos agresiva, de modo que se soluciona el error del algoritmo de Finkel.

El último algoritmo, propuesto hace tres años, computa el conjunto de mínima cobertura más rápidamente que los anteriores. A diferencia de ellos, el algoritmo MinCov guarda las aceleraciones, en lugar de los marcajes.

1.1. Objetivos y estructura del trabajo

El objetivo de este trabajo ha consistido en investigar, analizar y comparar los distintos algoritmos que se han desarrollado para computar el conjunto de mínima cobertura de una Red de Petri.

En cuanto a la estructura del trabajo, este comienza en el capítulo 3 con definiciones y nociones sobre las Redes de Petri. Del capítulo 4 al 8 se presentarán y analizarán los diferentes algoritmos que han sido objeto de estudio en este trabajo. Cada capítulo se centrará en un algoritmo específico, examinando sus principios de funcionamiento. Por último, en el capítulo 9 se resumirán las conclusiones clave derivadas del análisis y la comparación de

los algoritmos. Se destacarán las principales contribuciones y se ofrecerán recomendaciones prácticas basadas en los resultados obtenidos.

1.2. Plan de trabajo

El plan de trabajo ha sido el siguiente:

- Investigación preliminar sobre Redes de Petri y el conjunto de mínima cobertura.
- Estudio teórico detallado de cada algoritmo, comprendiendo su funcionamiento y aplicaciones.
- Comparación teórica de los algoritmos en función de su eficiencia y precisión.

Capítulo 2

Introduction

Over the last thirty years, several algorithms have been presented in order to compute the coverability set of Petri Nets. The importance of this set lies in the fact that it contains the markings that cover all the reachable markings of the Petri Net. Thus, it is possible to know the behaviour of the Petri Net, which of its places are bounded and which are not, which states are reachable and which are unreachable, etc.

In this paper we will present and analyse tha algorithms that have been developed to perform this task.

A starting point for the search of these algorithms was the Karp-Miller algorithm [1]. It computes a coverability tree that is not necessarily unique. To build the Karp-Miller tree of a Petri Net, we take its initial marking and assign it to the root node of the tree. From there and firing transitions, we reach new markings and making up the tree. This is based on the monotonicity of Petri Nets, which is a property that states that if starting from a marking M_1 and a transition t, we reach a marking M_1', then starting from a marking $M_2 \geq M_1$, it is possible to fire t from M_2', resulting in a marking $M_2' \geq M_1'$. To achieve this, accelerations are performed, in such a way that places in marking M_2 that have more tokens than those in marking M_1 are "saturated"to infinity, while leaving the other places intact. In this manner, an ω-marking has been constructed.

Finkel [1] used this algorithm to develop one that computes the minimum coverability tree of a Petri Net. Unlike the coverability tree, the minimum coverability tree is unique. In his algorithm, Finkel introduced a very aggressive pruning strategy which made tha algorithm incorrect, as shown only several years after, by a counterexample [6].

At this point there were two options for dealing with the Patri Nets coverability problem: try to fix Finke's algorithm or start from zero. The second option was what Raskin, Geeraerts and Van Begin did introducing the CovProc [2] algorithm. They defined the coverability sequence and worked with sets of pairs of markings. This innovative algorithm succeeded in correctly computing the minimal coverability set of Petri Nets.

From this point, attempts were made to find new algorithms that compute the minimal coverability set in a more efficient way. This is then than Piipponen and Valmari [3], Reynier and Servais [4], and finally Haddad, Finkel and Khmelnitsky [5] present three algorithms.

The first one, based on Tarjan's algorithm, follows the CovProc approach but uses sets of markings instead of pairs. On the other hand, the Monotonic-Pruning algorithm goes back to Finkel's initial idea but, unlike Finkel's, its pruning strategy is less agresive, so that the error of Finkel's algorithm is solved.

The latest algorithm, proposed three years ago, computes the minimal coverability set faster than the previous algorithms. Unike them, the MinCov algorithm stores the accelerations, instead of the markings.

2.1. Objectives and structure of the work

The aim of this work has been to investigate, analyze, and compare the different algorithms developed to compute the minimal coverability set of a Petri Net.

In terms of the work's structure it begins in Chapter 3 with definitions and concepts about Petri Nets. From Chapter 4 to 8, various algorithms studied in this work will be presented and analyzed. Each chapter will focus on an specific algorithm, examining its principles of operation. Finally, in Chapter 9, key conclusions derived from the analysis and comparison of the algorithms will be summarized. The main contributions will be highlighted, and practical recommendations based on the results obtained will be provided.

2.2. Work plan

The work plan has been as follows:

- Preliminary research on Petri Nets and minimal coverability set.
- Detailed theoretical study of each algorithm, understanding its operation and applications.
- Theoretical comparison of the algorithms based on their efficiency and accuracy.

Capítulo 3

Preliminares

3.1. Definiciones y notación

A lo largo de esta sección introductoria vamos a repasar conceptos básicos sobre las Redes de Petri que van a ser necesarios para abordar los algoritmos que se presentarán a lo largo de este texto.

Definición 3.1.1 (Red de Petri)**.** Una Red de Petri (*Petri Net*) es una tupla (P, T, W, M_0) donde $P = \{p_0, p_1, .., p_n\}$ es un conjunto finito de lugares (representados por círculos), $T = \{t_0, t_1, ..., t_m\}$ es un conjunto finito de transiciones (representadas por rectángulos), $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ son los pesos de los arcos y M_0 es el marcaje inicial de la red, esto es, la cantidad de marcas (*tokens*) que tiene cada lugar.

Dadas las precondiciones y postcondiciones de una transición t, estas son conjuntos de lugares de los que se toman y dejan los tokens, y se denotan respectivamente por ${}^\bullet t$ y $t^\bullet$; la transición t se puede lanzar si en cada $p \in {}^\bullet t$ se encuentran los tokens necesarios para ejecutar la transición y cada $p \in t^\bullet$ puede albergar los tokens que recibirá tras la ejecución, y entonces los tokens pasarán de los primeros lugares a los segundos.

Cuando una transición t se puede lanzar desde un marcaje M dando como resultado un nuevo marcaje M' escribimos $M \xrightarrow{t} M'$.

Definición 3.1.2 (Marcajes y ω-marcajes)**.**

- Un marcaje es una función $M : P \to \mathbb{N}$ y vamos a representarlo como un vector de la siguiente manera: $(M(p_0), M(p_1), ..., M(p_n))$.
- Un ω-marcaje es una aplicación $M : P \to \mathbb{N} \cup \{\omega\}$ donde $\omega \notin \mathbb{N}$

representa valores arbitrariamente grandes, que por ello se "saturan" como infinito.

Definición 3.1.3 (Cuasi-vivacidad)**.** Una transición t se dice cuasi-viva (*quasi-live*) si existe un marcaje alcanzable M de manera que t puede ser lanzada desde M. Esto es, t es cuasi-viva si puede ser lanzada desde al menos uno de los marcajes alcanzables de la red.

Definición 3.1.4 (Acotación)**.** Sea $k \in \mathbb{N}$ y p un lugar de una Red de Petri, se dice que p es k-acotado si para todo marcaje M se tiene que $M(p) \leq k$. Una Red de Petri se dice acotada si todos sus lugares están k-acotados.

Definición 3.1.5 (Deadlock)**.** Un marcaje se dice que es un marcaje muerto o deadlock si no se puede lanzar ninguna transición.

Proposición 3.1.1 (Monotonía)**.** *Sean M y M' marcajes alcanzables de una Red de Petri tales que $M \leq M'$ y t una transición ejecutable desde M y M'. Entonces t es ejecutable desde M'. Además, sean M_t y M'_t los marcajes tales que $M \xrightarrow{t} M_t$ y $M' \xrightarrow{t} M'_t$ entonces $M_t \leq M'_t$.*

Definición 3.1.6. Dado un conjunto P de lugares, un árbol etiquetado es una tupla $T = (N, B, r, \Lambda)$ tal que (N, B, r) es un árbol, N es el conjunto de sus nodos, B es el conjunto de sus vértices y $r \in N$ es el nodo raíz del árbol y $\Lambda : N \to (\mathbb{N} \cup \{\omega\})^{|P|}$ es la función que etiqueta los nodos con marcajes.

Sean $n, n' \in N$ se escribirá $B(n, n')$, $B^*(n, n')$ y $B^+(n, n')$ en lugar de $(n, n') \in B/B^*/B^+$.

Definición 3.1.7 (Conjunto y grafo de alcance)**.**

- El conjunto de alcance (*reachability set*) de una Red de Petri es el conjunto de los marcajes alcanzables desde el marcaje inicial M_0 de la Red.
- El grafo de alcance de una Red de Petri es un grafo dirigido y etiquetado de tal manera que sus nodos son los marcajes alcanzables y sus arcos están etiquetados con transiciones.

Definición 3.1.8 (Conjunto y grafo de cobertura)**.**

- El conjunto de cobertura (*coverability set*) de una Red de Petri es un conjunto de marcajes y ω-marcajes que cubren todos los marcajes alcanzables, de manera que para cada marcaje M' que está en el conjunto de cobertura, pero no en el de alcanzabilidad, existe una secuencia infinita estrictamente creciente de marcajes alcanzables $\{M_n\}_n$

que convergen a él, es decir, para todo lugar p y $n \in \mathbb{N}$, si $M'(p) \neq \omega$ entonces $M_n(p) = M'(p)$, en caso contrario $M_n(p) \geq n$, y lo escribimos $\lim M_n = M'$.

- El grafo de cobertura es un grafo dirigido que tiene por nodos los marcajes del conjunto de cobertura y cuyos arcos están etiquetados por transiciones.

La principal diferencia entre los árboles de alcanzabilidad y cobertura es que el de cobertura siempre va a ser finito, mientras que el de alcanzabilidad solo es finito si la Red de Petri está acotada.

3.2. El problema del conjunto de minima cobertura

En los sucesivos capítulos de esta memoria se presentan algoritmos que fueron propuestos para resolver el problema del conjunto de mínima cobertura.

Definición 3.2.1 (Conjunto y grafo de mínima cobertura)**.**

- Un conjunto es de mínima cobertura si ningún subconjunto propio de él es un conjunto de cobertura.
- El grafo de mínima cobertura de una Red de Petri es el único grafo de cobertura tal que su conjunto de nodos es el mínimo conjunto de cobertura.

Un conjunto mínimo de cobertura no puede contener dos marcajes comparables.

Lema 3.2.1 (Dickson)**.** *Sea S un conjunto finito y sean $\varphi_1, \varphi_2, ...$ una secuencia infinita de aplicaciones que van de S a $\mathbb{N} \cup \{\omega\}$, entonces existe una secuencia infinita de índices monótona y estrictamente creciente $i_1, i_2, ...$, tal que $\forall s \in S$ se tiene $\varphi_{i_1}(s) \leq \varphi_{i_2}(s) \leq ...$*

A continuación se presenta un resultado importante en relación a los conjuntos de mínima cobertura.

Lema 3.2.2. *El conjunto de mínima cobertura es finito y único.*

Demostración. 1. Supongamos que dicho conjunto es infinito. Entonces contendrá dos marcajes comparables $M < M'$. Pero esto no es posible y llegamos a una contradicción.

2. Supongamos ahora que hay dos conjuntos de mínima cobertura MC y MC', y sea $M \in MC \backslash MC'$. Distinguimos dos casos:

   a) Si M está en el conjunto de alcance R, por ser MC' un conjunto de cobertura, existe $M' \in MC'$ tal que $M < M'$. Si $M' \in R$ entonces existe $M'' \in MC$ tal que $M'' > M' > M$ de manera que M' es innecesario y MC no es mínimo. Si por el contrario $M' \notin R$ entonces, por el lema de Dickson, va a existir una secuencia infinita estrictamente creciente de marcajes alcanzables $\{M'_n\}_n$ que converge a M'. Como $M \in R$ para cierto índice q se tiene $M \leq M'_q$ y por ello MC no es mínimo.

   b) Si $M \notin R$ va a existir una secuencia infinita estrictamente creciente de marcajes alcanzables $\{M_n\}_n$ que converge a M. Existe $M' \in MC'$ tal que para todo n se tiene $M_n \leq M'$ y por tanto $M < M'$. Como $M \notin R$ y $M' > M$ necesariamente $M' \notin R$ con lo cual existe una secuencia $\{M'_n\}_n$ que converge a M'. Existe $M'' \in MC$ tal que para todo n se tiene $M'_n \leq M''$ y entonces $M' \leq M''$. Esto es una contradicción porque $M < M' \leq M''$ y $M, M'' \in MC$, y no puede haber dos marcajes comparables en un conjunto de mínima cobertura.

□

El problema de cobertura de las Redes de Petri es EXPSPACE-completo [7]. En primer lugar, se intentó abordar este problema modificando el algoritmo de Karp-Miller dando lugar al algoritmo del árbol de mínima cobertura (capítulo 4), pero se ha visto por medio de un contraejemplo que es erróneo y que solo computa una aproximación por debajo del mismo. Más tarde se desarrollaron diferentes algoritmos: CovProc (capítulo 5) que no está basado en el algoritmo de Karp-Miller e introduce la noción de secuencia de cobertura; el algoritmo StackProc (capítulo 6) y el de Poda-Monótona (capítulo 7) que desactiva nodos en lugar de eliminarlos. El algoritmo más eficiente es MinCov (capítulo 8), que toma como punto de partida el de Karp-Miller aplicando las nociones de abstracción y aceleración.

3.3. Algoritmo de Karp-Miller

Como punto de partida para el desarrollo de un algoritmo que computase el conjunto de mínima cobertura de una Red de Petri se tomó el algoritmo de Karp-Miller. Este computa un árbol de cobertura que no tiene por qué ser único.

3.3.1. Algoritmos del grafo y el árbol de Karp-Miller

El algoritmo de la figura 3.1 construye el árbol de Karp-Miller de una Red de Petri.

El procedimiento que sigue es el siguiente: mientras haya nodos sin procesar se toma uno para procesarlo. Para realizar este proceso se diferencian tres casos.

- Si hay un nodo n_1 anterior (*ancestor*) a n cuyo marcaje coincide con el de n, no tenemos que hacer nada.
- Si hay un nodo n_1 anterior a n de manera que $M_1 < M$ (donde M_1 es el marcaje de n_1 y M, el de n) convertimos M en un ω-marcaje de la siguiente manera: para cada lugar p tal que $M_1(p) < M(p)$ escribimos $M(p) = \omega$, en caso contrario se deja como estaba. Este proceso se denomina aceleración y por medio de ella se generan los ω-marcajes de manera rápida y eficiente.
- En otro caso, para cada transición t ejecutable desde M que dé como resultado el marcaje M' creamos un nuevo nodo n' cuyo marcaje es M', y un arco, (n, t, n'), donde $\Lambda(n) = M$ y $\Lambda(n') = M'$.

Supongamos que tenemos la sencilla red de Petri de la figura 3.2. Entonces sus respectivos árbol y grafo de Karp-Miller son los de las figuras 3.3 y 3.4.

Para la construcción del árbol se toma el marcaje inicial (1,0,0) y se ejecutan las distintas transiciones posibles para ir obteniendo marcajes que se añadirán al árbol. La primera aceleración llega con los marcajes $(0, 1, 0)$ y $(0, 1, 1)$, obteniéndose $(0, 1, \omega)$. La siguiente aceleración viene con este mismo marcaje y $(0, 2, \omega)$ que se obtiene al disparar t_3 desde $(0, 1, \omega)$. El marcaje acelerado en este caso es $(0, \omega, \omega)$. Desde este marcaje, disparando tanto t_2 como t_3, el marcaje resultante es él mismo. Así que detenemos el algoritmo, puesto que hemos terminado de computar el árbol de cobertura.

Sin embargo, el árbol de cobertura puede contener marcajes repetidos. Eliminaremos estas repeticiones creando un grafo a partir del árbol 3.3. Dado que desde $(0, 0, 1)$ disparando t_2 obtenemos el marcaje $(0, 1, 0)$ que ya habíamos alcanzado desde el marcaje inicial, eliminamos el subárbol que sigue al marcaje (0,0,1), que es el mismo que el de (0,1,0), y añadimos una flecha que conecta (0,0,1) y (0,1,0).

```
1  procedure Karp-Miller-Tree(PN) return KMT:
2  unprocesednodes := {create-node(r, M_0)}
3  while unprocessednodes ≠ ∅:
4    select node n ∈unprocessednodes
5    unprocessednodes := unprocessednodes-{n}
6    if ∃n_1 an ancestor node of n s.t. Λ(n) = Λ(n_1):
7      pass
8    if ∃n_1 an ancestor node of n s.t. Λ(n_1) < Λ(n):
9      M = Λ(n)
10     for all ancestors n_1 of n with Λ(n_1) < Λ(n) do
11       for all p ∈ P such that Λ(n_1)(p) < Λ(n)(p) do
12         M(p) := ω
13     Λ(n) := M
14     unprocessednodes := unprocessednodes+{n}
15   else:
16     for every t ∈ T such that Λ(n) -t-> Λ(n') do
17       create-node + arc((n, t, n'); KMT)
18       unprocessednodes := unprocessednodes+{n'}
```

Figura 3.1: Karp-Miller tree algorithm

De esta forma queda una representación mucho más compacta del conjunto de cobertura de la red.

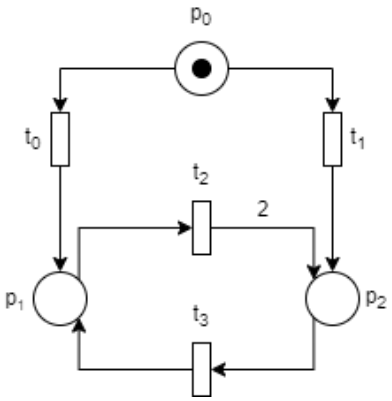

Figura 3.2: Ejemplo de Red de Petri

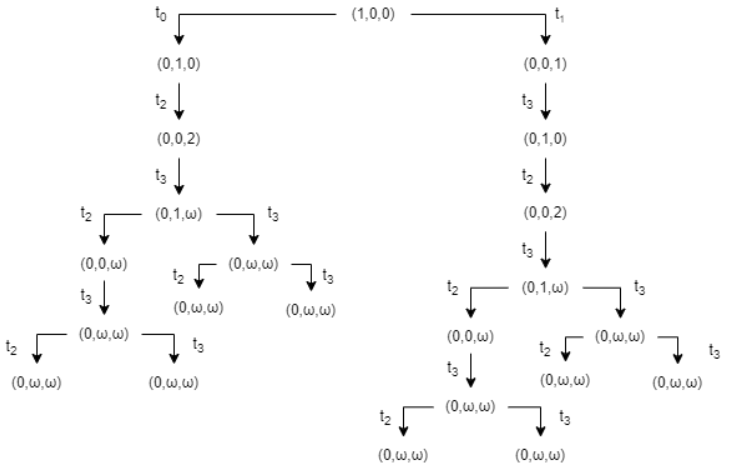

Figura 3.3: Árbol de Karp-Miller de la red de ejemplo

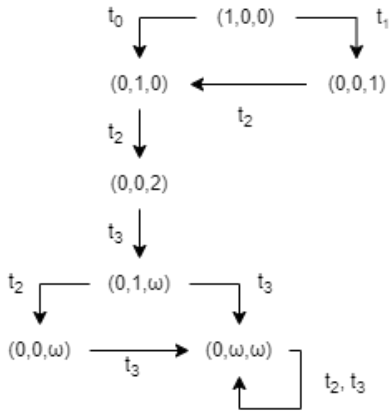

Figura 3.4: Grafo de Karp-Miller de la red de ejemplo

Capítulo 4

Algoritmo de Finkel

4.1. Motivación

Como ya hemos dicho anteriormente, Finkel se basó en el procedimiento de Karp-Miller y en el comportamiento monótono de las Redes de Petri, para desarrollar su algoritmo [1]. Las ideas en las que se basó para ello fueron las siguientes:

- Desarrollar el ábol de Karp-Miller hasta encontrar dos marcajes M y M' tales que $M \geq M'$. Entonces se poda el subárbol donde se encuentra el marcaje menor y se continúa explorando a partir de M.
- Compactar el anterior árbol de Karp-Miller reduciéndolo durante su desarrollo. Esto lo hacemos continuando con el subárbol del nodo etiquetado con el marcaje M' que no puede ser comparable con ninguno anterior M. Si por el contrario M y M' son comparables y se tiene que $M < M'$, computamos un nuevo marcaje M'' tal que para todo lugar p tal que $M'(p) > M(p)$ entonces $M''(p) = \omega$; en caso contrario tomamos $M''(p) = M'(p)$.
- Eliminar todo subárbol cuya raíz esté etiquetada con un marcaje M tal que $M < M''$.
- Identificar los nodos con la misma etiqueta, y mantener los arcos (M, t, M') de manera que t es ejecutable desde M y genera exactamente el marcaje M'.

4.2. Algoritmos del grafo y el árbol de mínima cobertura

Ejemplo 4.2.1. *El conjunto de mínima cobertura del ejemplo de red 3.2 es* $\{(1,0,0),(0,\omega,\omega)\}$*, pues todos los marcajes del árbol de cobertura de la red están cubiertos por ellos.*

Proposición 4.2.1. *El conjunto de etiquetas del grafo de Karp-Miller es un conjunto de cobertura finito y computable, y en general, no es el conjunto de mínima cobertura.*

Aunque sea posible computar el grafo de mínima cobertura a partir del grafo de Karp-Miller, la idea era construir dicho grafo directamente a partir de la Red de Petri, sin utilizar grafos intermedios. Para ello Finkel presentó un algoritmo [1] basado en la optimización del procedimiento de Karp-Miller. Se tuvieron en cuenta las siguientes ideas:

- Gracias a la monotonía de las Redes de Petri se puede parar una rama del árbol de Karp-Miller cuando encontramos dos nodos n_1 y n_2 con marcajes M_1 y M_2 tales que M_2 es alcanzable desde M_1 y $M_2 \leq M_1$. Con esto se sustituyen las líneas de código

```
if ∃n_1 an ancestor node of n s.t. Λ(n) = Λ(n_1):
    pass
```

  por las siguientes:

```
if ∃n_1 ∈processednodes s.t. Λ(n) = Λ(n_1):
    processednodes := processednodes + {n}
if ∃n_1 ∈processednodes s.t. Λ(n) < Λ(n_1):
    remove_node(n;MCT)
```

- Compactar el previamente reducido árbol de Karp-Miller durante su desarrollo. Para ello, continuamos con un marcaje siempre y cuando no sea comparable con uno anterior o si es estrictamente superior a uno de ellos. En este último caso, $M_2 > M_1$, para todos los lugares p tales que $M_2(p) > M_1(p)$ hacemos $M_2(p) = \omega$.

- Eliminar todo subárbol cuya raíz está etiquetada con $M_1 < M_2$.

```
1  procedure MC-Tree(PN) return MCT, MCS:
2  unprocesednodes := {create-node(r, M_0)}
3  processdednodes := ∅
4  while unprocessednodes≠ ∅:
5    select n ∈unprocessednodes
6    unprocessednodes := unprocessednodes - {n}
7    if ∃n_1 processed s.t. Λ(n) = Λ(n_1):
8       processednodes := processednodes+{n}
9    if ∃n_1 ∈ processednodes s.t. Λ(n) < Λ(n_1):
10      remove_node(n;MCT)
11   if ∃n_1 ∈ processednodes s.t. Λ(n_1) < Λ(n):
12      M := Λ(n), ancestor:=false
13      for all ancestors n_1 of n s.t. Λ(n_1) < Λ(n) do
14        for all p such that Λ(n_1)(p) < Λ(n)(p) do M(p) = ω
15      if ∃n_1 an ancestor of n s.t. Λ(n_1) < M then
16        ancestor:=true
17        n_1:=first node processed, on the path from
    the root to n s.t. Λ(n_1) < M
18        Λ(n_1) := M_2
19        remove_tree(n_1,MCT)
20        remove from (processed+unprocessednodes) all
     nodes of tree(n_1,MCT)
21        unprocessednodes := unporcessednodes + {n_1}
22      for every n_1 ∈processednodes s.t. Λ(n_1) < M do
23        remove from (processed+unprocessednodes) all
     nodes of tree(n_1,MCT)
24        remove_tree(n_1,MCT)
25        remove_node(n_1,MCT)
26      if ancestor=false then unprocessednodes :=
    unprocessednodes + {n}
27   else:
28      for every t s.t. Λ(n) --t--> Λ(n') do
29         create_node + arc((n,t,n');MCT)
30         unprocessednodes := unprocessednodes + {n'}
31      processednodes := processednodes+{n}
32 unprocessednodes:=maximal(unprocessednodes)
33 MCS:={label(n)|n ∈processednodes}
```

Figura 4.1: Algoritmo del árbol de mínima cobertura

- Tras identificar nodos con el mismo marcaje, dejar solo los arcos (n, t, n') tales que $\Lambda(n) \xrightarrow{t} \Lambda(n')$. Si por medio de exactamente una transición no se llega de un marcaje a otro, dejamos dichos marcajes sin unir.

Esta última idea no está incluida en el algoritmo y nos permitiría obtener el grafo de mínima cobertura y el conjunto de mínima cobertura, formado por los marcajes con los que se etiquetan los nodos del árbol. Dicho esto, el algoritmo computa un árbol de cobertura, que no es único ni de mínima cobertura, hasta que se le aplique el algoritmo 4.2.

```
1 procedure MC-Graph(PN) return MCG, MCS:
2  MCS, MCT := MC-Tree(PN)
3  identify_nodes_having_same_label(MCT,MCG)
4  for every arc(n,t,n') of MCG do
5    if not Λ(n) -t-> Λ(n') then remove arc((n,t,n',MCG))
```

Figura 4.2: Algoritmo del grafo de mínima cobertura

Volvamos con la Red de Petri el ejemplo 3.2 para obtener su conjunto de mínima cobertura. En primer lugar, figura 4.3(a), nos encontramos con que $(0, 0, 2) \geq (0, 0, 1)$, es decir, en una rama del árbol hay un marcaje comparable con uno de los marcajes de un nodo anterior. Eliminamos los nodos anteriores y pasamos al árbol de la figura 4.3(b). Como se puede ver, tenemos un ω-marcaje $(0, 1, \omega) \geq (0, 1, 0)$. Como antes, eliminamos el subárbol correspondiente e incorporamos nuevos marcajes. El árbol es ahora el de la figura 4.3(c). Como $(0, \omega, \omega) \geq (0, 1, \omega)$, repetimos el paso anterior. Tras eliminar los nodos anteriores y continuar con el árbol nos encontramos con marcajes repetidos (figura 4.3(d)), lo que significa que tenemos el árbol de mínima cobertura. Para obtener el grafo borramos los arcos que no son necesarios, obteniendo el mínimo grafo de cobertura en la figura 4.4.

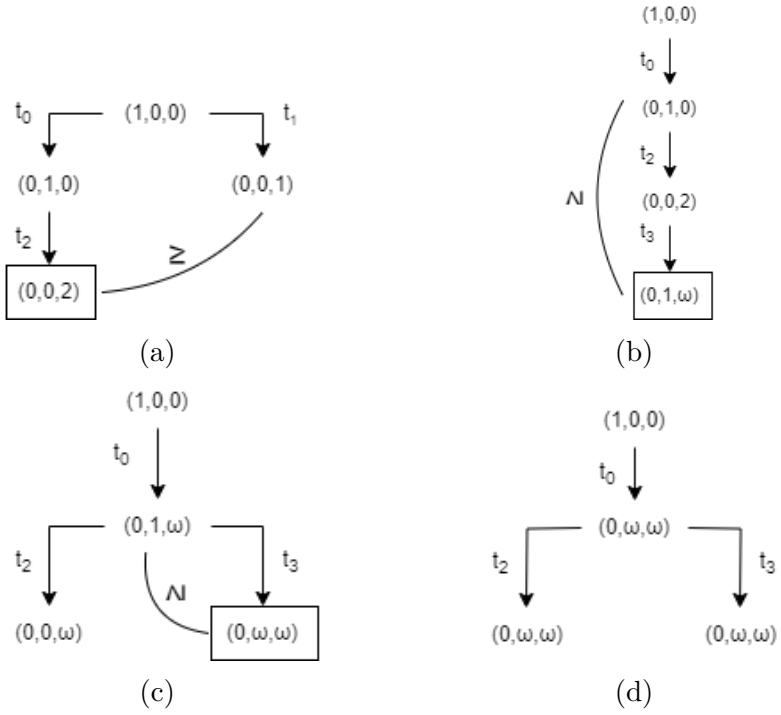

Figura 4.3: Obtención del grafo de mínima cobertura de la red de ejemplo

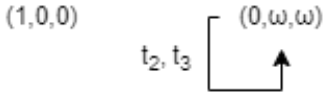

Figura 4.4: Conjunto de mínima cobertura de la red de ejemplo

4.3. Contraejemplo para el algoritmo

En 2005 se probó por medio de un contraejemplo [6] que este algoritmo no era correcto.

Observación 4.3.1. *Si el árbol de mínima cobertura tiene a lo sumo una ramificación, entonces el algoritmo de Finkel computará correctamente el árbol de mínima cobertura para dicha red. Sin embargo, si el árbol tiene dos o más ramificaciones el algoritmo no tiene por qué producir el árbol de mínima cobertura*

Imaginemos que tenemos un árbol como el de la figura 4.5(a), siendo α una secuencia de transiciones y $M_3 \geq M_2$. Entonces, se elimina la rama de la derecha y el árbol continúa a partir de M_3 (figura 4.5(b)). Por monotonía,

todos los marcajes, M, alcanzables desde M_2 van a ser cubiertos por todos los marcajes, M', alcanzables desde M_3; es decir, para cada M existirá un M' tal que $M \leq M'$.

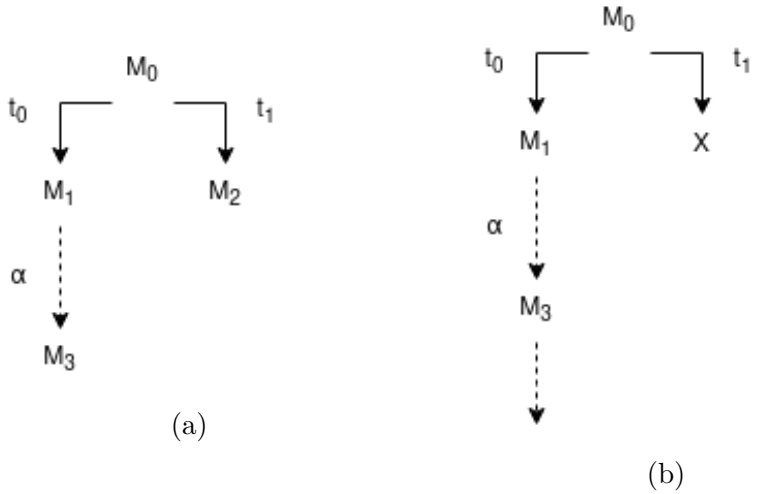

Figura 4.5

La red de Petri de la figura 4.6 constituye un contraejemplo al algoritmo de Finkel. Como se puede ver en el procedimiento de la figura 4.7, las tres ramas del árbol se cancelan entre ellas dando como resultado el árbol de la figura 4.8, que no es el árbol de mínima cobertura.

Como siempre, partimos del marcaje inicial y disparamos las posibles transiciones. Continuando con la segunda rama obtenemos el marcaje (0,0,0,0,1,0,1) que cubre a otro anterior en la misma rama: (0,0,0,0,1,0,0). Aceleramos el marcaje a (0,0,0,0,1,0,ω) y podamos la rama de manera que donde se encontraba (0,0,0,0,1,0,0) está ahora el ω-marcaje, y todos los descendientes del primero han sido eliminados (figura 4.7(a)).

Continuamos ahora con la rama de la izquierda (figura 4.7(b)) y llegamos al marcaje $(0,0,0,0,1,0,3) \leq (0,0,0,0,1,0,\omega)$, por lo que podamos la primera rama, de manera que no podemos continuar con ella.

Vamos ahora con la tercera rama. Llegamos al marcaje (0,0,1,0,0,0,1) que cubre el marcaje (0,0,1,0,0,0,0), por lo que podamos la segunda rama, pudiendo continuar con el árbol solo con la tercera rama (figura 4.7(c)). Así, siguiendo con esta rama llegamos al marcaje (0,0,0,0,0,1,1), que es menor que el marcaje (0,0,0,0,0,1,2) de la primera rama, así que podamos (figura

4.7(d)). Dado que no podemos continuar, el algoritmo termina.

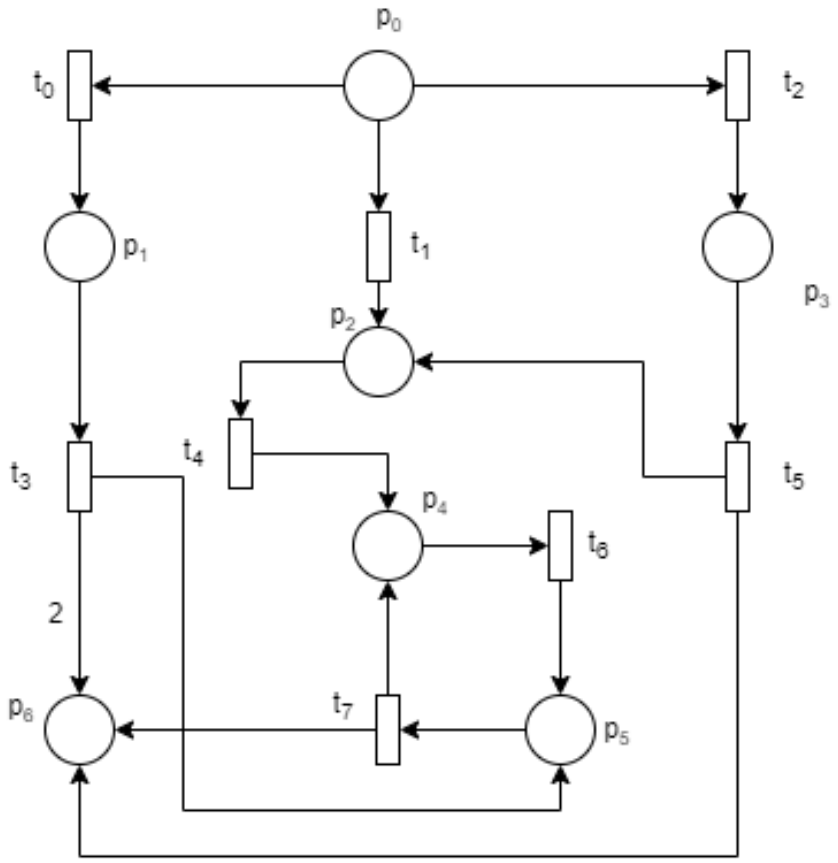

Figura 4.6: Contraejemplo al algoritmo de Finkel

De esta manera el árbol resultante es el de la figura 4.8, cuyos marcajes no son el conjunto de mínima cobertura, pues no cubren todos los marcajes alcanzables, como por ejemplo el (0,0,0,0,1,0,3).

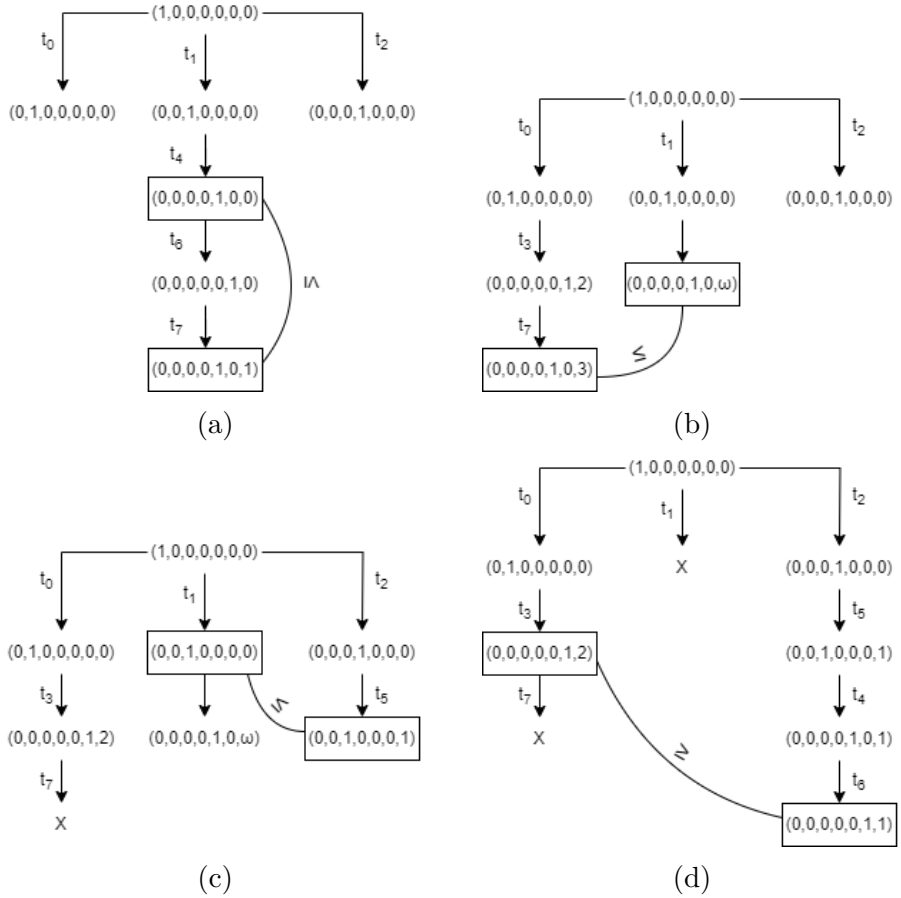

Figura 4.7

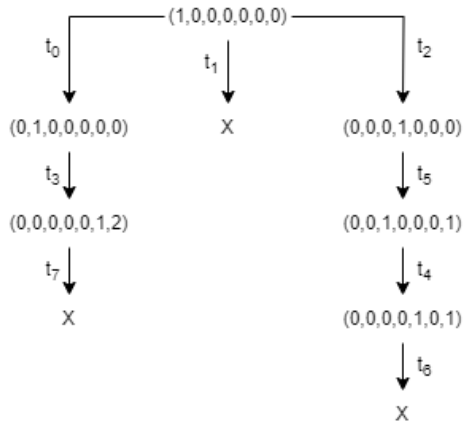

Figura 4.8

Capítulo 5

Algoritmo CovProc

5.1. Motivación

La principal idea del algoritmo anterior era construir un árbol donde todos los marcajes fueran incomparables. Esto se basaba en la función de aceleración de marcajes que se ve en la figura 5.1.

```
1 function acelerate(n):
2   foreach ancestor n_1 of n s.t. M_1 < M do
3     foreach p s.t. M_1(p) < M(p) do
4       M(p) = ω
5   return M
```

Figura 5.1

Así, los marcajes eliminados y los marcajes alcanzables a partir de ellos estarían cubiertos por los marcajes alcanzables que parten de ese marcaje que sirvió para eliminar un subárbol. Eso sería así gracias a la monotonía de las Redes de Petri. Ya vimos con un contraejemplo que, aunque la idea parecía buena, el algoritmo falla en algún caso no dando como resultado el árbol de mínima cobertura, aunque sí una aproximación del mismo.

En lugar de intentar arreglar el algoritmo de Finkel, Raskin, Geeraerts y Van Begin parten de cero con la intención de obtener un método eficiente (pues el de Finkel tampoco lo era mucho) para obtener el conjunto de mínima cobertura. El algoritmo [2] se basa en una serie de novedosas ideas:

- En lugar de construir un árbol, se va a trabajar con conjuntos de pares de marcajes.

- Se fijará una forma para ordenar los marcajes en los pares, para poder explotar la propiedad de monotonía de las Redes de Petri.

De este modo se busca trabajar con conjuntos que sean mínimos.

En primer lugar, dado un marcaje M definimos el conjunto $\text{Post}(M) = \{M' | \exists t \in T : M \xrightarrow{t} M'\}$. De esta manera se tiene que $\text{Post}^*(M)$ es el conjunto de marcajes alcanzables desde M.

Definición 5.1.1. Dado M un ω-marcaje, definimos $[\![M]\!] = \{M_n \in \mathbb{N}^P \mid M_n \leq M\}$ y lo extendemos a conjuntos de ω-marcajes.

Se quiere computar una representación efectiva de $[\![\text{Post}^*(M_0)]\!]$, que es el límite de la secuencia infinita $X_0 = [\![M_0]\!]$ y $X_i = [\![X_{i-1} \cup \text{Post}(X_{i-1})]\!]$ para $i \geq 1$. Gracias a la monotonía de las Redes de Petri se puede considerar la secuencia $Y_0 = \text{Max}^{\preccurlyeq}(\{M_0\})$ e $Y_i = \text{Max}^{\preccurlyeq}(Y_{i-1} \cup \text{Post}(Y_{i-1}))$ para $i \geq 1$.

5.2. La secuencia de cobertura

Para trabajar con pares de marcajes se van a introducir las siguientes nociones:

- Sea $R \subseteq P \times P$, $\text{Flatten}(R)$ es el conjunto de los segundos marcajes de todos los pares de marcajes de R, es decir, $\text{Flatten}(R) = \{M | \exists M' : (M', M) \in R\}$.

- $\overline{\text{Post}}$ es la función para definir el sucesor de un par de marcajes. $\overline{\text{Post}}((M_1, M_2)) = \{(M_1, M'), (M_2, M') | M' \in \text{Post}(M_2)\}$.

- Dados dos marcajes $M_1 \leq M_2$, decimos que $\text{AccelPair}(M_1, M_2) = M_\omega$ si para todo $p \in P$, si $M_1(p) = M_2(p)$ entonces $M_\omega(p) = M_1(p)$, y en caso contrario ($M_1(p) < M_2(p)$), $M_\omega(p) = \omega$.

- $\overline{\text{Accel}}$ va a ser la función que se utilice para definir la noción de aceleración de un par de marcajes tales que $M_1 < M_2$. En este caso $\overline{\text{Accel}}((M_1, M_2)) = \{(M_2, \text{AccelPair}(M_1, M_2))\}$. En caso contrario la función no está definida.

- Las funciones $\overline{\text{Post}}$ y $\overline{\text{Accel}}$ se extienden a conjuntos de pares de marcajes, R, tomando

$$\overline{\text{Post}}(R) = \cup_{(M_1,M_2)\in R} \overline{\text{Post}}((M_1, M_2))$$

$$\overline{\text{Accel}}(R) = \cup_{(M_1,M_2)\in R, M_1<M_2} \overline{\text{Accel}}((M_1, M_2))$$

Definición 5.2.1. Sean M_1 y M_2 dos marcajes, se define la función $M_1 \ominus M_2 : P \to \mathbb{Z} \cup \{-\omega, \omega\}$ como

$$(M_1 \ominus M_2)(p) = \begin{cases} \omega, & \text{si } M_1(p) = \omega \\ -\omega, & \text{si } M_2(p) = \omega \text{ y } M_1(p) \neq \omega \\ M_1(p) - M_2(p), & \text{en otro caso} \end{cases}$$

Así, dados dos pares de marcajes (M_1, M_2) y (M_1', M_2'), tenemos que $(M_1, M_2) \sqsubseteq (M_1', M_2')$ si $M_1 \leq M_1'$, $M_2 \leq M_2'$ y, para todo $p \in P$, $(M_1 \ominus M_2)(p) \leq (M_1' \ominus M_2')(p)$.

Para todo par (M_1, M_2) y conjunto de pares de marcajes R, se tienen

- ${}^{\sqsubseteq}[\![(M_1, M_2)]\!] = \{(M_1', M_2') | (M_1', M_2') \sqsubseteq (M_1, M_2)\}$
- ${}^{\sqsubseteq}[\![R]\!] = \cup_{(M_1, M_2) \in R} {}^{\preccurlyeq}[\![(M_1, M_2)]\!]$
- $\text{Max}^{\sqsubseteq}(R) = \{(M_1, M_2) \in R | \nexists (M_1', M_2') \in R : (M_1, M_2) \neq (M_1', M_2') \wedge (M_1, M_2) \sqsubseteq (M_1', M_2')\}$

Lema 5.2.1. *Sean M_1 y M_2 dos marcajes de una Red de Petri tales que $M_1 \leq M_2$ y ${}^{\preccurlyeq}[\![M_2]\!] \subseteq {}^{\preccurlyeq}[\![Post^*(M_1)]\!]$, entonces se tiene que*

$${}^{\preccurlyeq}[\![AccelPair((M_1, M_2))]\!] \subseteq {}^{\preccurlyeq}[\![Post^*(M_2)]\!]$$

Demostración. Sea $P' = \{p \in P | M_1(p) < M_2(p)\}$, como $M_1 \leq M_2$ entonces $P \setminus P' = \{p \in P | M_1(p) = M_2(p)\}$. Como $M_1 \leq M_2$ y ${}^{\preccurlyeq}[\![M_2]\!] \subseteq {}^{\preccurlyeq}[\![\text{Post}^*(M_1)]\!]$, existe una secuencia de transiciones σ lanzable desde M_1 que permite aumentar el número de tokens de los lugares de P'.

Sea $\overline{M}$ tal que $M_1 \xrightarrow{\sigma} \overline{M}$, entonces $M_1 \leq \overline{M}$ y $M_1(p) < \overline{M}(p)$ para todo $p \in P'$.

Sea $\overline{M}_i$, para $i \geq 1$, el marcaje tal que $M_1 \xrightarrow{\sigma^i} \overline{M}_i$ tenemos que

$$\forall i \geq 1 : \forall p : \overline{M}_i(p) = M_1(p) + i(\overline{M}(p) - M_1(p))$$

Para todo $i \geq 1$, $\overline{M}_i \in \text{Post}^*(M_1)$ y $\overline{M}_i \leq \overline{M}_{i+1}$. Para $p \in P'$, $\overline{M}(p) - M_1(p) > 0$ así que

$$\forall p \in P' : \forall n \in \mathbb{N} : \exists k : \overline{M}_k(p) > n$$

$$\forall p \in P \setminus P' : \forall i \geq 1 : \overline{M}_i(p) \geq M_1(p) = M_2(p) = \text{AccelPair}(M_1, M_2)(p)$$

Consideremos $M \in {}^{\preccurlyeq}[\![\text{AccelPair}((M_1, M_2))]\!]$. Entonces $M \leq \text{AccelPair}((M_1, M_2))$ y para todo $p \in P$ $M(p) \neq \omega$. Para todo $p \in P'$ y para todo $i \geq 1$,

$M(p) \leq \overline{M}_i(p)$. Además, para todo $p \in P'$ existe k_p tal que $\overline{M}_{k_p}(p) > M(p)$. Como la secuencia $\{\overline{M}_i\}_{i\geq 1}$ es creciente y $\overline{M}_i \leq \overline{M}_{i+1}$ para todo $i \geq 1$, el marcaje $\overline{M}_k$ donde $k = \text{máx}\{k_p | p \in P'\}$ es tal que $\forall p \in P' : \overline{M}_k(p) \geq M(p)$, concluyendo así que $\overline{M}_k \geq M$.

Como $\overline{M}_k \in \text{Post}^*(M_1)$ y $M_1 \leq M_2$, por monotonía existe un marcaje M' de manera que $M' \in \text{Post}^*(M_2)$ y $M \leq \overline{M}_k \leq M'$. Como $M \in {}^{\preccurlyeq}[\![\text{AccelPair}((M_1, M_2))]\!]$, concluimos que ${}^{\preccurlyeq}[\![\text{AccelPair}((M_1, M_2))]\!] \subseteq {}^{\preccurlyeq}[\![\text{Post}^*(M_2)]\!]$. □

La idea en la que se basa el algoritmo es la siguiente: contaríamos con dos conjuntos V y F, de pares visitados y pares que constituyen la frontera, respectivamente, de manera que en cada iteración vamos almacenando los pares de marcajes que cubren a los vistos en el paso anterior en V y F, en el conjunto V, y descubrimos con $\overline{\text{Post}}$ y $\overline{\text{Accel}}$ nuevos pares que almacenamos en F. Los pares de marcajes que se guardan en V se eliminan de F.

Sin embargo, y lo veremos con un ejemplo, este algoritmo no es lo suficientemente eficiente pues en los conjuntos V se van a ir almacenando una gran cantidad de pares de marcajes, y en ningún momento se elimina un par. Para evitar esto se introduce el siguiente concepto:

Definición 5.2.2 (Oráculo)**.** Dada una Red de Petri (P, T, W, M_0), un oráculo es una función $\text{Oracle} : \mathbb{N} \to P((\mathbb{N} \cup \{\omega\})^{|P|} \times (\mathbb{N} \cup \{\omega\})^{|P|})$ que, para todo $i \geq 0$, devuelve un conjunto de pares de marcajes que satisfacen las dos siguientes condiciones:

$$ {}^{\preccurlyeq}[\![\text{Post}(\text{Flatten}(\text{Oracle}(i)))]\!] \subseteq {}^{\preccurlyeq}[\![\text{Flatten}(\text{Oracle}(i))]\!] $$

$$ {}^{\preccurlyeq}[\![\text{Flatten}(\text{Oracle}(i))]\!] \subseteq \mathit{Cover}(PN) $$

donde $\mathit{Cover}(PN)$ es el conjunto de cobertura de la red. De esta manera, para todo $i \geq 0$, $\text{Post}(\text{Flatten}(\text{Oracle}(i))) \subseteq \text{Flatten}(\text{Oracle}(i))$ y $\text{Flatten}(\text{Oracle}(i))$ es una aproximación por abajo del conjunto de cobertura de la Red de Petri.

Con el oráculo y unos conjuntos V y F de pares de marcajes tenemos la definición de la secuencia de cobertura en la que se basará el algoritmo.

Definición 5.2.3 (Secuencia de cobertura)**.** Sean $PN = (P, T, W, M_0)$ una Red de Petri y un oráculo, Oracle, la secuencia de cobertura de la red, denotada como CovSeq(PN,M_0,Oracle), es la secuencia infinita $(V_i, F_i, O_i)_{i\geq 0}$, definida de la siguiente manera:

- $V_0 = \emptyset$, $O_0 = \emptyset$ y $F_0 = \{(M_0, M_0)\}$

- Para todo $i \geq 1$: $O_i := \mathrm{Max}^{\sqsubseteq}(O_{i-1} \cup \mathrm{Oracle}(i))$
- Para todo $i \geq 1$: $V_i := \mathrm{Max}^{\sqsubseteq}(V_{i-1} \cup F_{i-1}) \setminus {}^{\preccurlyeq}[\![O_i]\!]$
- Para todo $i \geq 1$: $F_i := \mathrm{Max}^{\sqsubseteq}(\overline{\mathrm{Post}}(F_{i-1}) \cup \overline{\mathrm{Accel}}(F_{i-1})) \setminus {}^{\preccurlyeq}[\![O_i \cup V_i]\!]$

Es decir, una secuencia de cobertura es una secuencia de tuplas (V_i, F_i, O_i) donde O_i es el conjunto de pares que representan la información del oráculo, V_i es el conjunto que representa los pares visitados y F_i es el conjunto de pares que representan la frontera.

Para construir la secuencia de cobertura se parte de una versión simplificada de la misma, usando el oráculo vacío dado por $Oracle(i) = \emptyset$, $\forall i \geq 0$.

En primer lugar, se considera $(V_i, F_i)_{i\geq 0}$ donde $F_0 = \{(M_0, M_0)\}$, $V_0 = \emptyset$ y, para todo $i \geq 1$: $V_i = V_{i-1} \cup F_{i-1}$ y $F_i = \overline{Post}(F_{i-1}) \setminus V_i$. Así, esta secuencia consiste en computar, paso a paso, todos los sucesores del marcaje inicial, en orden ascendente.

En segundo lugar, se considera la secuencia $(V_i, F_i)_{i\geq 0}$ donde ahora para todo $i \geq 1$ se tiene $F_i = (\overline{\mathrm{Post}}(F_{i-1}) \cup \overline{\mathrm{Accel}}(F_i - 1)) \setminus V_i$.

Después, explotamos el orden $\sqsubseteq$ tomando para todo $i \geq 1$: $V_i = \mathrm{Max}^{\sqsubseteq}(V_{i-1} \cup F_{i-1})$ y $F_i = \mathrm{Max}^{\sqsubseteq}(\overline{\mathrm{Post}}(F_{i-1}) \cup \overline{\mathrm{Accel}}(F_{i-1})) \setminus {}^{\sqsubseteq}[\![V_i]\!]$.

Finalmente, asumimos que el oráculo ya no va a ser trivial y consideramos la secuencia de cobertura de la definición 5.2.3. En ella almacenamos los pares devueltos por el oráculo en los conjuntos O_i, eliminaremos de V_i y F_i los pares cubiertos por algún O_i.

Probaremos que existe $k \geq 0$ tal que ${}^{\preccurlyeq}[\![\mathrm{Flatten}(O_k \cup V_k)]\!]$ es exactamente el conjunto de cobertura.

5.2.1. Invariantes de la secuencia de cobertura

Los siguientes lemas constituyen varios invariantes de la secuencia de cobertura que resultan indispensables para probar la corrección del algoritmo.

Lema 5.2.2. *Sean A y B dos conjuntos de marcajes de una red de Petri, entonces*

$$ {}^{\preccurlyeq}[\![A]\!] \subseteq {}^{\preccurlyeq}[\![Post^*(B)]\!] \Rightarrow {}^{\preccurlyeq}[\![Post^*(A)]\!] \subseteq {}^{\preccurlyeq}[\![Post^*(B)]\!] $$

Lema 5.2.3. *Sean PN una Red de Petri con marcaje inicial M_0 , Oracle, un oráculo y CovSeq(PN,M_0,Oracle) = $(V_i, F_i, O_i)_{i\geq 0}$; entonces*

1. $\forall i \geq 0 : \overline{Post}(V_i) \cup \overline{Accel}(V_i) \subseteq {}^{\sqsubseteq}[\![V_i \cup F_i \cup O_i]\!]$
2. $\forall i \geq 0 : {}^{\preccurlyeq}[\![Flatten(V_i \cup O_i)]\!] \subseteq {}^{\preccurlyeq}[\![Flatten(V_{i+1} \cup O_{i+1})]\!]$
3. $\forall i \geq 0 : \forall (M_1, M_2) \in F_i \cup V_i : {}^{\preccurlyeq}[\![M_2]\!] \subseteq {}^{\preccurlyeq}[\![Post^*(M_1)]\!]$
4. $\forall i \geq 0 : {}^{\preccurlyeq}[\![Post^*(Flatten(O_i))]\!] \subseteq {}^{\preccurlyeq}[\![Flatten(O_i)]\!]$

Gracias a estos invariantes podemos demostrar que la secuencia de cobertura permite obtener el conjunto de cobertura de una Red de Petri. Para ello primero hay que probar que la secuencia es completa en el sentido de que existe k tal que ${}^{\preccurlyeq}[\![\text{Flatten}(V_k)]\!]$ es el conjunto de cobertura de la Red de Petri; y después, probar que la secuencia no computará marcajes que no pertenecen al conjunto de cobertura de la Red de Petri, y por tanto es correcta.

5.2.2. Completitud de la secuencia de cobertura

Se probará ahora que la secuencia de cobertura es correcta. Pero antes hay que probar que cada marcaje obtenido por el algoritmo de Karp-Miller está cubierto por algún marcaje calculado por la secuencia de cobertura. Para ello, se necesitan dos lemas.

Lema 5.2.4. *Sean PN una Red de Petri con marcaje inicial M_0 , Oracle, un oráculo y CovSeq(PN,M_0,Oracle) = $(V_i, F_i, O_i)_{i\geq 0}$. Dados $i \geq 0$ y $M \in Flatten(V_i)$, y sea $\sigma = t_1 t_2 ... t_l$ una secuencia no vacía de transiciones ejecutable desde M de manera que $M_1, M_2, ..., M_l$ son los marcajes tales que $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} ... \xrightarrow{t_l} M_l$. Entonces existe k con $i+1 \leq k \leq i+l$ tal que*

- *O bien hay un par $(\widehat{M}, \widehat{M_l}) \in V_k$ con $(M, M_l) \sqsubseteq (\widehat{M}, \widehat{M_l})$;*
- *o bien ${}^{\preccurlyeq}[\![M_l]\!] \subseteq {}^{\preccurlyeq}[\![Flatten(O_k)]\!]$.*

Definición 5.2.4. Sean PN una Red de Petri con marcaje inicial M_0 y $T = (N, B, r, \Lambda)$ su árbol de Karp-Miller, entonces $\varsigma : N \to T^*$ es la función que asocia una secuencia de transiciones a cada nodo de la siguiente manera:

- Si n es la raíz del árbol o no existe un antecesor n' de n tal que $\Lambda(n') < \Lambda(n)$ entonces $\varsigma(n)$ devuelve la secuencia vacía.
- En otro caso, sean los conjuntos de lugares $P_\Lambda = \{p \in P | \Lambda(n)(p) = \omega \text{ y } M(n)(p) \neq \omega\}$ y $P_M = \{p \in P | \Lambda(n)(p) = M(n)(p) = \omega\}$, $\varsigma(n)$ devuelve la secuencia tal que para todo $p \in P_\Lambda : \varsigma(n)(p) > 0$; para todo $p \in P \setminus (P_\Lambda \cup P_M) : \varsigma(n)(p) = 0$ y $\varsigma(n)$ es ejecutable desde $M(n)$.

donde $M(n)$ es el marcaje tal que $\Lambda(n)$ se ha obtenido por medio de su aceleración.

Lema 5.2.5. *Sean PN una Red de Petri con marcaje inicial M_0, Oracle, un oráculo, $T = (N, B, r, \Lambda)$ el árbol de Karp-Miller de la red de Petri y CovSeq(PN,M_0,Oracle) $= (V_i, F_i, O_i)_{i\geq 0}$, entonces:*

$$\forall n \in N : \forall k \geq \sum_{n' \ antecesor \ de \ n} (|\varsigma(n')| + 2) : \quad {}^{\preccurlyeq}[\![\Lambda(n)]\!] \subseteq {}^{\preccurlyeq}[\![Flatten(V_k \cup O_k)]\!]$$

Como consecuencia se tiene que la secuencia de cobertura es completa.

Corolario 5.2.1. *Sean PN una Red de Petri con marcaje inicial M_0, Oracle, un oráculo y CovSeq(PN,M_0,Oracle) $= (V_i, F_i, O_i)_{i\geq 0}$; entonces existe $k \geq 0$ tal que para todo $l \geq k$, $Cover(PN, M_0) \subseteq {}^{\preccurlyeq}[\![Flatten(V_l \cup O_l)]\!]$, donde $Cover(PN, M_0)$ es el conjunto de cobertura de la Red de Petri.*

5.2.3. Corrección de la secuencia de cobertura

Para probar que la secuencia es correcta hay que ver que todo marcaje M producido por ella cumple ${}^{\preccurlyeq}[\![M]\!] \subseteq \text{Cover}(PN, M_0)$.

Lema 5.2.6. *Sean PN una Red de Petri con marcaje inicial M_0 , Oracle, un oráculo y CovSeq(PN,M_0,Oracle) $= (V_i, F_i, O_i)_{i\geq 0}$. Entonces*

$$\forall i \geq 1 : \forall M \in Flatten(V_i \cup F_i \cup O_i) : \quad {}^{\preccurlyeq}[\![M]\!] \subseteq {}^{\preccurlyeq}[\![Post^*(M_0)]\!]$$

Como consecuencia de este lema se tiene el siguiente resultado.

Corolario 5.2.2. *Sean PN una Red de Petri con marcaje inicial M_0 , Oracle, un oráculo y CovSeq(PN,M_0,Oracle) $= (V_i, F_i, O_i)_{i\geq 0}$. Entonces, $\forall i \geq 1$, ${}^{\preccurlyeq}[\![Flatten(V_i \cup O_i)]\!] \subseteq Cover(PN, M_0)$.*

Este corolario y el 5.2.1 permiten obtener el siguiente teorema.

Teorema 5.2.1. *Sean PN una Red de Petri con marcaje inicial M_0 , Oracle, un oráculo y CovSeq(PN,M_0,Oracle) $= (V_i, F_i, O_i)_{i\geq 0}$. Entonces, existe $k \geq 0$ tal que*

- $\forall 1 \leq i < k : \quad {}^{\preccurlyeq}[\![Flatten(V_i \cup O_i)]\!] \subset {}^{\preccurlyeq}[\![Flatten(V_{i+1} \cup O_{i+1})]\!]$;
- $\forall i \geq k : \quad {}^{\preccurlyeq}[\![Flatten(V_i \cup O_i)]\!] = Cover(PN, M_0)$.

5.3. Algoritmo

Para llevar todo esto a la práctica es importante contar con un oráculo optimizado. Lo definiremos de manera recursiva sobre los marcajes resultantes de una aceleración, dando prioridad a los marcajes M para los cuales la mayoría de lugares verifican $M(p) = \omega$.

```
1  procedure CovProc(PN, M_0):
2      i := 0
3      O_0 := ∅
4      V_0 := ∅
5      F_0 := {(M_0, M_0)}
6      repeat
7          i := i + 1
8          R_i := ∪_{M∈S} CovProc(PN, M)  where  S = Flatten(Accel(F_{i-1}))
9          O_i := Max^⊑(O_{i-1} ∪ R_i)
10         V_i := Max^⊑(V_{i-1} ∪ F_{i-1}) \ ≼⟦O_i⟧
11         F_i := Max^⊑(Post(F_{i-1}) ∪ Accel(F_{i-1})) \ ≼⟦O_i ∪ V_i⟧
12     until ≼⟦Flatten(O_i ∪ V_i)⟧ ⊆ ≼⟦Flatten(O_{i-1} ∪ V_{i-1})⟧
13     return (O_i ∪ V_i)
```

Figura 5.2

Para cada paso i, el oráculo se ha implementado como un número finito de llamadas recursivas a CovProc, donde los marcajes iniciales son el resultado de acelerar los marcajes obtenidos en dicho paso.

El algoritmo es correcto y lo vamos a probar por medio del teorema siguiente.

Teorema 5.3.1. *Para toda Red de Petri, PN, y para todo marcaje, M_0, $CovProc(PN, M_0)$ termina y $\preccurlyeq[\![Flatten(CovProc(PN, M_0))]\!] = Cover(PN, M_0)$.*

Demostración. La demostración es por inducción sobre $N_\omega(M_0) = |\{p \in P : M(p) = \omega\}|$.

Si $N_\omega(M_0) = |P|$, entonces CovProc(PN, M_0) termina tras dos iteraciones y devuelve $\overline{O}_2 \cup \overline{V}_2 = \{(M_0, M_0)\}$. No se efectúa ninguna otra llamada recursiva, pues $R_1 = R_2 = \emptyset$. Además, $\preccurlyeq[\![\text{Flatten}(\text{CovProc}(PN, M_0))]\!] = \preccurlyeq[\![M_0]\!] = \text{Cover}(PN, M_0)$.

Si por el contrario $N_\omega(M_0) = k < |P|$, se consideran dos casos:

- El algoritmo termina tras l iteraciones, esto es ${}^{\preccurlyeq}[\![\text{Flatten}(\overline{O}_l \cup \overline{V}_l)]\!] = {}^{\preccurlyeq}[\![\text{Flatten}(\overline{O}_{l-1} \cup \overline{V}_{l-1})]\!]$ y para todo $1 \leq j \leq l-1$: ${}^{\preccurlyeq}[\![\text{Flatten}(\overline{O}_j \cup \overline{V}_j)]\!] \neq {}^{\preccurlyeq}[\![\text{Flatten}(\overline{O}_{j-1} \cup \overline{V}_{j-1})]\!]$. Si para todo $1 \leq j \leq l$, y para todo $M \in \text{Flatten}(\overline{F}_j) : N_\omega(M) \geq k$, entonces para todo $M' \in \text{Flatten}(\overline{\text{Accel}}(\overline{F}_j)) : \quad N_\omega(M') \geq k+1$ y $\text{CovProc}(PN, M')$ termina. Así, ${}^{\preccurlyeq}[\![\text{Flatten}(\text{CovProc}(PN, M'))]\!] = \text{Cover}(PN, M')$. Se concluye que R_j está computado en tiempo finito y ${}^{\preccurlyeq}[\![\text{Post}(\text{Flatten}(R_j))]\!] \subseteq {}^{\preccurlyeq}[\![\text{Flatten}(R_j)]\!] \subseteq \text{Cover}(PN, M_0)$, para todo $1 \leq j \leq l$.

  Por el teorema 5.2.1., existe $k \geq 0$ tal que

  - $\forall 1 \leq i < k : \quad {}^{\preccurlyeq}[\![Flatten(V_i \cup O_i)]\!] \subset {}^{\preccurlyeq}[\![Flatten(V_{i+1} \cup O_{i+1})]\!]$;
  - $\forall i \geq k : \quad {}^{\preccurlyeq}[\![Flatten(V_i \cup O_i)]\!] = Cover(PN, M_0)$.

  Así, $k = l$ y concluimos que el algoritmo termina y devuelve $\text{Cover}(PN, M_0)$.

- Supongamos que el procedimiento no termina. Esto puede ser porque el bucle se repite infinitamente, o porque en algún paso j el bucle tarde un tiempo infinito en completarse. Esto último no es posible porque R_j se calcula en tiempo finito y las funciones $\text{Max}^{\sqsubseteq}$, Flatten, Post y Accel son computables. De modo que si el algoritmo no termina entonces computa una secuencia infinita $(V_i, F_i, O_i)_{i \geq 0}$. Como $\{R_j\}_{j \geq 0}$ es finito, para todo $j \geq 0$ se tiene $V_j = \overline{V}_j$ y $O_j = \overline{O}_j$. Nuevamente por el teorema 5.2.1. concluimos que existe $k \geq 1$ tal que ${}^{\preccurlyeq}[\![\text{Flatten}(\overline{V}_k \cup \overline{O}_k)]\!] = {}^{\preccurlyeq}[\![\text{Flatten}(\overline{V}_{k-1} \cup \overline{O}_{k-1})]\!]$. Con lo cual el algoritmo terminaría teniéndose una contradicción.

□

Ejemplo 5.3.1. *Asumamos que el oráculo es trivial, es decir, vacío. Veamos cómo se aplica este algoritmo a la Red de Petri 3.2. Primero se inicializan* $V_0 = \emptyset$ *y* $F_0 = \{((1,0,0),(1,0,0))\}$.

Paso 1. Incluimos en V_1 *el par de* F_0*. En cuanto a* F_1*, dado que no podemos acelerar el par ((1,0,0),(1,0,0)), hacemos* $\overline{Post}((1,0,0),(1,0,0)) = \{((1,0,0), M), ((1,0,0), M') \mid \exists t : (1,0,0) \xrightarrow{t} M \wedge (1,0,0) \xrightarrow{t} M'\} = \{((1,0,0), M) \mid \exists t : (1,0,0) \xrightarrow{t} M\} = \{((1,0,0),(0,1,0)),((1,0,0),(0,0,1))\}$*, pues (0,1,0) y (0,0,1) se obtienen al disparar* t_0 *y* t_1 *desde (1,0,0). Además eliminamos de* F_1 *el par ((1,0,0),(1,0,0)).*

- $V_1 = \{((1,0,0),(1,0,0))\}$
- $F_1 = \{((1,0,0),(0,0,1)),((1,0,0),(0,1,0))\}$

Paso 2. Incluimos en V_2 los pares de marcajes de F_1. Nuevamente no podemos acelerar los pares de F_1, así que exploramos nuevos marcajes como antes: $\overline{Post}((1,0,0),(0,1,0)) = \{((1,0,0),(0,0,2)),((0,1,0),(0,0,2))\}$ *y* $\overline{Post}((1,0,0),(0,0,1)) = \{((1,0,0),(0,1,0)),((0,0,1),(0,1,0))\}$. *Incluimos estos nuevos pares en F_2 y eliminamos los que se encuentran también en V_2. Como* $((1,0,0),(0,0,1)) \sqsubseteq ((1,0,0),(0,0,2))$, *eliminamos el par menor de F_2.*

- $V_2 = \{((1,0,0),(1,0,0)),((1,0,0),(0,0,1)),((1,0,0),(0,1,0))\}$
- $F_2 = \{((1,0,0),(0,0,2)),((0,1,0),(0,0,2)),((0,0,1),(0,1,0))\}$

Paso 3. El procedimiento es el mismo que en el paso anterior. Podemos ver que tenemos dos pares en F_3, ((0,1,0),(0,1,1)) y ((0,0,1),(0,0,2)), sobre los que vamos a poder aplicar una aceleración en el paso cuarto.

- $V_3 = \{((1,0,0),(1,0,0)),((1,0,0),(0,0,2)),((1,0,0),(0,1,0)),$
  $((0,1,0),(0,0,2)),((0,0,1),(0,1,0))\}$
- $F_3 = \{((1,0,0),(0,1,1)),((0,0,2),(0,1,1)),((0,1,0),(0,1,1)),$
  $((0,0,1),(0,0,2))\}$

Paso 4. Añadimos a V_4 los pares de F_3 y eliminamos el par ((1,0,0),(0,1,0)), ya que $((1,0,0),(0,1,0)) \sqsubseteq ((1,0,0),(0,1,1))$. *En F_4 aceleramos los pares ((0,1,0),(0,1,1)) y ((0,0,1),(0,0,2)) obteniendo* $((0,1,1),(0,1,\omega))$ *y* $((0,0,2),(0,0,\omega))$, *además de seguir descubriendo marcajes por medio de $\overline{Post}$.*

- $V_4 = \{((1,0,0),(1,0,0)),((1,0,0),(0,0,2)),((1,0,0),(0,1,1)),$
  $((0,1,0),(0,0,2)),((0,1,0),(0,1,1)),((0,0,1),(0,0,2)),((0,0,1),(0,1,1))\}$
- $F_4 = \{((1,0,0),(0,0,3)),((1,0,0),(0,2,0)),((0,1,1),(0,2,0)),$
  $((0,0,2),(0,2,0)),((0,1,1),(0,1,\omega)),((0,0,2),(0,1,1)),((0,0,2),(0,0,\omega))\}$

Paso 5. Volvemos a añadir a V_5 los pares de F_4 y eliminamos los que sean cubiertos por otros. En F_5 no podemos acelerar pares así que exploramos nuevos marcajes. Al eliminar pares redundantes del conjunto V_4 queda un conjunto F_5 formado por un par donde solo podemos efectuar $\overline{Post}$ y otro más que aceleraremos en el siguiente paso del algoritmo.

- $V_5 = \{((1,0,0),(1,0,0)),((1,0,0),(0,1,1)),((1,0,0),(0,2,0)),$
  $((1,0,0),(0,0,3)),((0,1,1),(0,2,0)),((0,1,1),(0,1,\omega)),((0,0,2),(0,2,0)),$
  $((0,0,2),(0,1,1)),((0,0,2),(0,0,\omega))\}$

- $F_5 = \{((1,0,0),(0,1,2)),((0,1,\omega),(0,2,\omega))\}$

Paso 6. Incluimos en V_6 los dos pares de F_5 y eliminamos pares que están cubiertos por otros. En F_6 aceleramos el par $((0,1,\omega),(0,2,\omega))$ obteniendo $((0,2,\omega),(0,\omega,\omega))$. Aplicamos $\overline{Post}$ al par restante y obtenemos los pares $((1,0,0),(0,2,1))$ y $((0,1,2),(0,2,1))$. Pero solo nos quedamos con el primero, pues $((0,1,2),(0,2,1)) \sqsubseteq ((0,2,\omega),(0,\omega,\omega))$.

- $V_6 = \{((1,0,0),(1,0,0)),((1,0,0),(0,1,2)),((1,0,0),(0,2,0)),$
  $((1,0,0),(0,0,3)),((0,1,\omega),(0,2,\omega))\}$
- $F_6 = \{((1,0,0),(0,2,1)),((0,2,\omega),(0,\omega,\omega))\}$

Paso 7.

- $V_7 = \{((1,0,0),(1,0,0)),((1,0,0),(0,1,2)),((1,0,0),(0,0,3)),$
  $((1,0,0),(0,2,0)),((0,2,\omega),(0,\omega,\omega))\}$
- $F_7 = \{((1,0,0),(0,1,3)),((1,0,0),(0,3,0)),((0,\omega,\omega),(0,\omega,\omega))\}$

Paso 8.

- $V_8 = \{((1,0,0),(1,0,0)),((1,0,0),(0,1,3)),((1,0,0),(0,3,0)),$
  $((0,\omega,\omega),(0,\omega,\omega))\}$
- $F_8 = \{((1,0,0),(0,0,5)),((1,0,0),(0,2,2)),((0,\omega,\omega),(0,\omega,\omega))\}$

Se tiene que ${}^{\preccurlyeq}[\![Flatten(V_7)]\!] = \{(1,0,0),(0,\omega,\omega)\} = {}^{\preccurlyeq}[\![Flatten(V_8)]\!]$. Con lo cual, hemos obtenido el conjunto de mínima cobertura de la Red de Petri, que es $\{(1,0,0),(0,\omega,\omega)\}$.

Aunque la red es muy sencilla, con este marcaje inicial, los conjuntos de pares de marcajes tienen ya bastantes elementos. Vamos arrastrando pares ((1,0,0),M) en V donde $M \leq (0,\omega,\omega)$, y, por tanto, M no va a incluirse en el conjunto de mínima cobertura. Pero estos pares no se pueden eliminar de V, pues los pares (M_1, M_2) donde M_2 es un ω-marcaje, son incomparables con los pares ((1,0,0),M). Así, estos pares se van a ir arrastrando hasta el final del algoritmo. Es por esto la importancia de tener un oráculo eficiente. Aquí hemos tomado el vacío porque se trata de una red sencilla y estamos ejecutando el algoritmo a mano, pero si utilizáramos un oráculo eficiente los conjuntos V no deberían contener los pares $((1,0,0),M)$, que no aportan nada a la computación del conjunto de mínima cobertura.

Si ahora contamos con el oráculo definido en el algoritmo, el procedimiento

sería el siguiente:

Paso 0. Inicialmente $O_0 = \emptyset$, $V_0 = \emptyset$ *y* $F_0 = \{((1,0,0),(1,0,0))\}$.

Paso 1. $V_1 = \{((1,0,0),(1,0,0))\}$ *y* $F_1 = \{((1,0,0),(0,0,1)),((1,0,0),(0,1,0))\}$. *Como no se ha producido ninguna aceleración en* F_1 *el oráculo sigue siendo vacío.*

Paso 2. Como en el paso anterior, el oráculo sigue siendo vacío, así $V_2 = \{((1,0,0),(1,0,0)),((1,0,0),(0,0,1)),((1,0,0),(0,1,0))\}$ *y* $F_2 = \{((1,0,0),(0,0,2)),((0,1,0),(0,0,2)),((0,0,1),(0,1,0))\}$.

Nuevamente, en el paso 3 el oráculo es también vacío, por lo que $V_3 = \{((1,0,0),(1,0,0)),((1,0,0),(0,0,2)),((1,0,0),(0,1,0)),((0,1,0),(0,0,2)),((0,0,1),(0,1,0))\}$ *y* $F_3 = \{((1,0,0),(0,1,1)),((0,0,2),(0,1,1)),((0,1,0),(0,1,1)),((0,0,1),(0,0,2))\}$

En el paso 4 tendremos un oráculo no vacío, pues en F_3 *hay un par, ((0,0,1),(0,0,2)) que se puede acelerar a* $((0,0,2),(0,0,\omega))$. *Así,* $O_4 := Max^{\sqsubseteq}(O_3 \cup CovProc(PN,(0,0,\omega)))$. *Con los cálculos que hicimos antes en la aplicación del algoritmo sin oráculo es fácil ver que* $O_4 = \{((0,\omega,\omega),(0,\omega,\omega))\}$. *De esta manera* $V_4 = \{((1,0,0),(1,0,0)),((1,0,0),(0,0,2)),((1,0,0),(0,1,1))\}$ *mientras que* $F_4 = \{((1,0,0),(0,0,3)),((1,0,0),(0,2,0)),((0,1,1),(0,1,\omega)),((0,0,2),(0,1,1)),((0,0,2),(0,0,\omega))\}$. *Como se puede ver, se ha reducido el tamaño de estos dos conjuntos, sin perderse por ello información importante.*

Paso 5. Nuevamente $O_5 = \{((0,\omega,\omega),(0,\omega,\omega))\}$, *por lo que* $V_5 = \{((1,0,0),(1,0,0)),((1,0,0),(0,1,1)),((1,0,0),(0,2,0)),((1,0,0),(0,0,3))\}$ *y* $F_5 = \{((1,0,0),(0,1,2)),((0,1,\omega),(0,2,\omega))\}$.

Podemos ver que $^{\preccurlyeq}[\![Flatten(O_4 \cup V_4)]\!] = \{((1,0,0),(1,0,0)),((0,\omega,\omega),(0,\omega,\omega))\} = {}^{\preccurlyeq}[\![Flatten(O_5 \cup V_5)]\!]$. *Con lo cual el algoritmo termina en 5 pasos, en lugar de en 8, como necesitábamos cuando no usamos el oráculo.*

Capítulo 6

Algoritmo de StackProc

6.1. Motivación

Pocos años más tarde, Piipponen y Valmari desarrollan el algoritmo [3]. Este sigue la lógica del algoritmo anterior, a diferencia de que se trabaja con marcajes en lugar de pares de marcajes. Durante la ejecución del mismo se guardan los marcajes visitados y los que cubren a estos. La principal novedad que presenta este algoritmo es que se van a almacenar los marcajes a partir de los cuales se explorarán nuevos. Esto se hará por medio de una pila que guarda los marcajes y las transiciones que se han de disparar desde ellos. De esta manera se quiere reducir el gasto en memoria con respecto al algoritmo anterior.

Aparte de esta pila, se tienen dos conjuntos donde se almacenarán los marcajes y cada marcaje tiene un atributo tr que apunta a la siguiente transición que debe intentar dispararse desde dicho marcaje.

La construcción del conjunto de mínima cobertura va a basarse en las siguientes ideas:

- Partir de un marcaje M_0 de modo que si σ es una secuencia de transiciones y M es el marcaje resultante del disparo $M_0 \xrightarrow{\sigma} M$, se puede disparar una transición t desde M. Así, σt se puede ejecutar repetidamente obteniendo M' donde para ciertos lugares $M'(p) = M_0(p)$ y para otros, $M'(p)$ crece sin límite. El límite de tokens de estos últimos lugares va a ser ω, por tanto construimos un marcaje M'' de manera que si $M'(p) > M_0(p)$ entonces $M''(p) = \omega$, mientras que $M''(p) = M_0(p)$ cuando $M'(p) = M_0(p)$.

- En lugar de $M_0 \xrightarrow{\sigma} M$, se utilizará $M_0 \xrightarrow{t}{}^{\omega} M$.
- Tras el disparo de t en M, se debe usar más de un M_0 y un σ para tener $M_0 \xrightarrow{\sigma}{}^{\omega} M$ y añadir ω.
- No hace falta recordar las transiciones tales que $M \xrightarrow{t}{}^{\omega} M''$, solo que existen tales transiciones.

Definición 6.1.1 (Condición de bombeo)**.** Decimos que se cumple la condición de bombeo si $M_0 < M'$ y existen $\sigma \in T^*$ tal que $M_0 \xrightarrow{\sigma}{}^{\omega} M'$ y $p \in P$ tal que $M_0(p) < M'(p) < \omega$. La operación de bombeo consiste en asignar ω a aquellos $M'(p)$ para los cuales $M_0(p) < M'(p) < \omega$.

6.2. Algoritmo

Dos conjuntos de marcajes que tienen un papel importante en el algoritmo son los siguientes:

- El conjunto A contiene los marcajes maximales que se han encontrado hasta el momento.
- El conjunto F es una tabla hash. Los marcajes se añaden a F al mismo tiempo que al conjunto A, con la diferencia de que nunca se eliminarán de F, aunque sí de A, cuando se encuentre un marcaje que cubra a otro. Así, $A \subseteq F$.

Escribimos $M \xrightarrow{t}{}^{\omega} M'$ para expresar que $M \xrightarrow{t} M''$ con $M'' > M$ y o bien $M'' \in F$, y en tal caso $M'' = M'$, o bien M'' ha sido transformado en M' cambiando $M''(p)$ a ω cuando corresponda con la operación de bombeo (esto se verá con CoverCheck) y añadiendo M' a F a continuación.

Definición 6.2.1. Una componente fuertemente conexa de un grafo dirigido (V, E) es un conjunto maximal de vértices $V' \subseteq V$ tal que $\forall u, v \in V'$ hay un camino de u a v.

En nuestro caso, V es el conjunto de todos los ω-marcajes que se han encontrado y no se han descartado durante la construcción del conjunto de mínima cobertura.

Además de W, en el algoritmo se utiliza otra pila, S, cuyos elementos también son punteros a los de F. Cada ω-marcaje encontrado se introduce en S.

Todo marcaje tiene dos atributos: el índice (*index*), que es el número invariable que recibe el marcaje cuando se encuentra por primera vez, y el *lowlink*, que es el menor índice de cualquier marcaje que se sepa que pertenece a la misma componente fuerte que el marcaje actual. Todo marcaje que se añade a S es eliminado de ese conjunto cuando se encuentra una componente fuertemente conexa del mismo.

En primer lugar, se inicializa el algoritmo añadiendo el marcaje inicial a F, A, W y S. Después, se prueba a disparar cada transición t en cada ω-marcaje M encontrado. Si dicha transición no se puede ejecutar, se rechaza; en caso contrario, se dispara. Si el marcaje alcanzado ya se había encontrado antes, se rechaza.

CoverCheck(M', A) comprueba si M' está cubierto por algún marcaje de A. Si es así, descartamos M' y si no, comprueba si la condición de bombeo se cumple para cualquier $M_0 \in A$. Si es así, cambia $M'(p)$ a ω cuando corresponda, lo que hace que M' crezca estrictamente y que la condición de bombeo se cumpla para otro $M_0' \in A$. CoverCheck también elimina aquellos marcajes M de A que sean cubiertos por M' y cuando esto ocurre, M.tr se pone a cero para que si el algoritmo vuelve a ese marcaje no se disparen desde él transiciones. CoverCheck continúa mientras no haya un marcaje $M_0 \in A$ que verifique la condición de bombeo.

Lema 6.2.1. *Para todo $M_0 \in F$ hay un marcaje $M_0' \in A$ tal que $M_0 \leq M_0'$.*

Esto se debe a que cada vez que un marcaje se inserta en F, también se añade en A; y cuando un marcaje M_0 se elimina de A es porque un marcaje $M_0' \geq M_0$ se añade en A. Este lema garantiza que todos los ω-marcajes que pueden ser alcanzados disparando transiciones desde cierto marcaje van a ser cubiertos por los ω-marcajes que obtiene el algoritmo. A continuación vamos a probar esto, para lo que serán necesarios una serie de lemas.

Definición 6.2.2. Sea M_c el marcaje actual del algoritmo, un ω-marcaje M está recién-madurado si $M \in A$ y si el algoritmo ha terminado o para algún $p \in P$ se tiene que $M_c(p) < M(p) = \omega$. M ha madurado si está o ha estado recién-madurado.

El siguiente lema se tiene por construcción del algoritmo, pues hasta que no se ha intentado ejecutar todas las transiciones de un marcaje, este no termina.

Lema 6.2.2. *Si M ha madurado, entonces el algoritmo ha intentado ejecutar todas las transiciones en M.*

```
1  procedure StackProc(PN):
2   F := {M_0}; A := {M_0}; W.push(M_0); M_0.tr := first transition
3   S.push(M_0); M_0.ready := false; n_f := 1; M_0.index := 1;
      M_0.lowlink := 1
4   while W ≠ ∅ do
5    M := W.top; t := M.tr
6    if t ≠ nil then M.tr := next transition
7    if t = nil then
8      W.pop
9      activate transitions
10     if M.lowlink:= M.index then
11       while S.top ≠ W.top do
12         S.top.ready:=true; S.pop
13     else
14       W.top.lowlink:=min{W.top.lowlink,M.lowlink}
15   if ¬M[t⟩ then go to line 3
16   M' :=the marking s.t. M -t-> M'
17   if M' ≤ M then passivate t and go to line 4
18   if M' ∈ F then
19     if ¬M'.ready then M.lowlink:=min{M.lowlink,M'.
      lowlink} and go to line 4
20   CoverCheck(M', A)
21   if M' is covered by an elem of A then go to line 4
22   F := F ∪ {M'}; A := A ∪ {M'}; W.push(M'); M'.tr:=first
      transition
23   S.push(M'); M'.ready:=false
24   n_f := n_f + 1; M'.index:= n_f; M'.lowlink:= n_f
```

Figura 6.1: Algoritmo StackProc

Lema 6.2.3. *Si M'_0 está recién-madurado, $M'_0 \geq M_0$ y $M_0 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$, entonces para $1 \leq i \leq n$ hay un marcaje recién-madurado M'_i tal que $M_i \leq M'_i$ y $M'_i \geq M'_{i-1} + \Delta_{t_i} \geq M'_0 + \Delta_{t_1 \ldots t_i}$.*

Lema 6.2.4. *Si M'_0 está recién-madurado, $M'_0 \geq M_0$ y $M_0 \xrightarrow{t_1}{}^{\omega} M_2 \xrightarrow{t_2}{}^{\omega} \ldots \xrightarrow{t_n}{}^{\omega} M_n$, entonces para $1 \leq i \leq n$ hay un marcaje recién-madurado M'_i tal que $M_i \leq M'_i$ y $M'_i \geq M'_{i-1} + \Delta_{t_i} \geq M'_0 + \Delta_{t_1 \ldots t_i}$.*

De estos lemas se obtiene el siguiente corolario.

Corolario 6.2.1. *Si M'_0 ha madurado, $M'_0 \geq M_0$ y $M_0 \xrightarrow{t_1} M_2 \xrightarrow{t_2} ... \xrightarrow{t_n} M_n$ o $M_0 \xrightarrow{t_1} {}^{\omega}M_2 \xrightarrow{t_2} {}^{\omega}... \xrightarrow{t_n} {}^{\omega}M_n$, entonces para $1 \leq i \leq n$ existe $M'_i \in F$ madurado tal que $M_i \leq M'_i$ y $M'_i \geq M'_{i-1} + \Delta_{t_i} \geq M'_0 + \Delta_{t_1...t_i}$, por lo que en adelante el algoritmo no va a intentar ejecutar transiciones en ningún $M''_i \leq M'_i$.*

Gracias a este corolario, el algoritmo evitará explorar marcajes que no contribuyan a la construcción del conjunto de mínima cobertura.

Si al disparar una transición el marcaje resultante es menor o igual que el de partida, dicha transición es inútil. En el algoritmo, "primera transición" y "siguiente transición" se seleccionan las transiciones de una lista doblemente enlazada que llamaremos lista activa. La línea 16 del algoritmo comprueba si la transición actual se ha vuelto inútil. Si es así, se elimina de la lista y se añade a una lista "pasiva", $passive[c]$. La transición t se ignora hasta que el algoritmo retrocede a un ω-marcaje cercano al cual se añadieron símbolos ω, y todas las transiciones de la lista $passive[c]$ se eliminan y añaden a la lista activa.

Veamos ahora un ejemplo práctico.

Ejemplo 6.2.1. *Queremos obtener nuevamente el conjunto de mínima cobertura de la Red de Petri 3.2. Procederemos de la siguiente manera. Primero inicializamos $F = \{(1,0,0)\}$, $A = \{(1,0,0)\}$, $W = \{(1,0,0)\}$ y $S = \{(1,0,0)\}$. Así mismo $M_0.index = 1$, $M_0.lowlink = 1$ y $n_f = 1$.*

Paso 1. $M = (1,0,0)$, $t = t_0$ y $M' = (0,1,0)$. Entonces $A = \{(1,0,0),(0,1,0)\}$, $F = \{(1,0,0),(0,1,0)\}$, $S = \{(1,0,0),(0,1,0)\}$ y $W = \{(1,0,0),(0,1,0)\}$. Además, $n_f = 2$, $M'.index = 2$ y $M'.lowlink = 2$.

Paso 2. Seguimos con $M = (1,0,0)$, ejecutamos la siguiente transición, $t = t_1$, obteniendo $M' = (0,0,1)$. Actualizamos, $A = \{(1,0,0),(0,1,0),(0,0,1)\}$, $F = \{(1,0,0),(0,1,0),(0,0,1)\}$, $S = \{(1,0,0),(0,1,0),(0,0,1)\}$ y $W = \{(1,0,0),(0,1,0),(0,0,1)\}$. En cuanto a los índices, $n_f = 3$ y $M'.index = M'.lowlink = 3$.

Paso 3. Desde $M = (1,0,0)$ no podemos ejecutar las siguientes transiciones, así que lo eliminamos de S y W, y pasamos al siguiente marcaje $M = (0,1,0)$. No podemos disparar t_0 ni t_1 pero sí t_2 y eso hacemos obteniendo $M' = (0,0,2)$. Como en A tenemos el marcaje $(0,0,1)$ que está cubierto por M', hacemos $M' = (0,0,\omega)$. Actualizamos, $A = \{(1,0,0),(0,1,0),(0,0,\omega)\}$,

$F = \{(1,0,0),(0,1,0),(0,0,1),(0,0,\omega)\}$, $S = \{(0,1,0),(0,0,1),(0,0,\omega)\}$ *y* $W = \{(0,1,0),(0,0,1),(0,0,\omega)\}$. *Por otro lado,* $n_f = 4$, *así que* $M'.lowlink = M'.index = 4$.

Paso 4. $M = (0,0,1)$ *pues desde* $(0,1,0)$ *no podemos disparar las siguientes transiciones, y* $t = t_3$. *Obtenemos* $M' = (0,1,1)$, *pero como* $(0,1,0) \in A$, *entonces* $M' = (0,1,\omega)$. *Así,* $A = \{(1,0,0),(0,1,\omega)\}$, $F = \{(1,0,0),(0,1,0),(0,0,1),(0,0,\omega),(0,1,\omega)\}$, $S = \{(0,0,1),(0,0,\omega),(0,1,\omega)\}$ *y* $W = \{(0,0,1),(0,0,\omega),(0,1,\omega)\}$. *Además,* $n_f = 5$, $M'.lowlink = M'.index = 5$.

Paso 5. No se pueden disparar las siguientes transiciones desde $(0,0,1)$ *así que pasamos al marcaje* $M = (0,0,\omega)$ *y a la transición* $t = t_3$, *obteniendo* $M' = (0,1,\omega)$. *Como* $(0,0,\omega) \in A$ *hacemos* $M' = (0,\omega,\omega)$. *Actualizamos,* $A = \{(1,0,0),(0,\omega,\omega)\}$, $F = \{(1,0,0),(0,1,0),(0,0,1),(0,0,\omega),(0,1,\omega),(0,\omega,\omega)\}$, $S = \{(0,0,\omega),(0,1,\omega),(0,\omega,\omega)\}$ *y* $W = \{(0,0,\omega),(0,1,\omega),(0,\omega,\omega)\}$. $n_f = 6$ *y los índices son* $M'.index = 6$ *y* $M'.lowlink = 6$.

Paso 6. Como no podemos ejecutar las siguientes transiciones desde $(0,0,\omega)$ *tomamos* $M = (0,1,\omega)$. *Para* $t = t_2$, $M' = (0,0,\omega)$ *que ya está cubierto por un marcaje de* A *y además está en* F. *Por ello,* $M.lowlink = \min\{M.lowlink, M'.lowlink\} = \min\{5,4\} = 4$. *Entonces disparamos* $t = t_3$ *obteniendo* $M' = (0,2,\omega)$ *que está cubierto nuevamente por un marcaje de* A.

Paso 7. Ahora $M = (0,\omega,\omega)$. *Disparando tanto* t_2 *como* t_3 *vamos a obtener* M. *En este punto* $W = \emptyset$, *por lo que hemos terminado. El conjunto de mínima cobertura es* $\{(1,0,0),(0,\omega,\omega)\}$.

Capítulo 7

Algoritmo de Poda-Monótona

7.1. Motivación

En este capítulo introducimos el algoritmo propuesto por Reynier y Servais de Poda-Monótona [4] [8], una versión mejorada del algoritmo de Karp-Miller, que tiene una estrategia de poda ligeramente menos agresiva que el algoritmo de Finkel.

Este algoritmo se sirve de una función de aceleración que toma un conjunto de ω-marcajes y un ω-marcaje y devuelve un ω-marcaje que puede ser usado para remplazar el ω-marcaje dado. Una función de aceleración clásica es la siguiente:

$$\mathrm{Acc}_{\mathrm{all}}(\mathcal{M}, M)(p) = \begin{cases} \omega & \text{si } \exists M' \in \mathcal{M} | M' < M \wedge M'(p) < M(p) < \omega \\ M(p) & \text{en caso contrario} \end{cases}, \forall p \in P$$

Una función de aceleración más débil, $\mathrm{Acc}_{\mathrm{one}}$ sería como la anterior quitando la condición $M' < M$.

7.2. Algoritmo

Al igual que el algoritmo de Karp-Miller, el de Poda-Monótona (figura 7.1) devuelve un árbol con los nodos etiquetados con marcajes y los arcos, con transiciones, y un conjunto Act, donde Λ(Act) es el conjunto de mínima cobertura de la Red de Petri. La función de aceleración se utiliza para alcanzar el marcaje límite. A diferencia del algoritmo de Finkel, este nuevo algoritmo no puede podar ramas que estén cubiertos por nodos de otras, de modo que no se eliminan nodos que sí son necesarios para construir el árbol

de cobertura. Los nodos del árbol se dividen en activos e inactivos. Los nodos activos formarán el conjunto de mínima cobertura, y los inactivos se mantendrán para garantizar la completitud del algoritmo.

Dado un par (n, t) eliminado de Wait, se introduce el nuevo nodo obtenido de este par en $\mathbf{C}$ de la siguiente manera:

1. Se computa el marcaje sucesor $M = \text{Post}(\Lambda(n), t)$.
2. n tiene que estar activo y M no debe estar cubierto por un nodo activo.
3. El marcaje resultante de la aceleración de M es computado y se asocia con un nuevo nodo n'.
4. Ciertos nodos se eliminan del conjunto Act de activos cuando se encuentran marcajes que cubren los de esos nodos.
5. n' se declara activo y se actualiza Wait. Sea un nodo n^* tal que $\Lambda(n^*) \leq \Lambda(n')$, entonces n^* se puede desactivar de dos maneras: si $n^* \notin \text{Ancestor}_{\mathcal{C}}(n')$ entonces todos sus descendientes se desactivan, y si, por el contrario $n^* \in \text{Ancestor}_{\mathbf{C}}(n')$ todos sus descendientes se desactivan excepto n', que se añade a Act.

```
1  procedure Monotonic-Pruning(PN, Acc):
2    x_0 :=new node s.t. Λ(x_0) = M_0
3    X := {x_0}, Act:= X, Wait:= {(x_0, t)|Λ(x_0) -t-> ·}, B := ∅
4    while Wait≠ ∅ do
5      Pop(n, t) from Wait, M := Post(Λ(n), t)
6      if n ∈Act and M ≰ Λ(Act) then
7        n' := new node s.t. Λ(n') = Acc(Λ(Ancestor_C(n) ∩ Act), M)
8        X := X ∪ {n'}, B := B ∪ {(n, t, n')}
9        Act := Act \ {x|∃y ∈ Ancestor_C(x) s.t.
   Λ(y) ≤ Λ(n') ∧ (y ∈ Act ∨ y ∉ Ancestor_C(n'))}
10       Act :=Act∪{n'}, Wait :=Wait∪{(n', t')|Λ(n') -t'-> ·}
11   return C = (X, B, x_0, Λ), Act
```

Figura 7.1: Algoritmo de Poda-Monótona

7.2.1. Corrección del algoritmo

Decimos que el algoritmo de Poda-Monótona es coherente si $\forall M \in \text{Act}$ existe M' en el conjunto de mínima cobertura de la red tal que $M \leq M'$; y completo si para todo M' del conjunto de mínima cobertura existe $M \in \text{Act}$ tal que $M' \leq M$. Si el algoritmo es coherente y completo entonces es correcto: computa el conjunto de mínima cobertura de la red.

Veamos primero que el algoritmo termina:

Teorema 7.2.1. *El algoritmo de Poda-Monótona termina.*

Demostración. Supongamos lo contrario, entonces **C** es un árbol infinito y por el lema de Dickson, contiene al menos una rama infinita $b = x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} x_2...$ con $(x_i, t_i, x_{i+1}) \in B$. Sea $n \in X \setminus \{x_i \mid i \geq 0\}$. Afirmamos que n no puede desactivar ningún x_i. Por contradicción, si n desactivara cierto x_i, entonces desactivaría todos sus descendientes a excepción de n en el caso de que fuera uno de ellos, que no lo es por definición. Pero esta desactivación no es posible dado que b es infinita.

Por definición de la función de aceleración, pueden darse dos casos: que aparezca un nuevo ω en el marcaje resultante y que no. Si se da el primer caso, todos los descendientes de dicho marcaje tendrán ese mismo ω. Como la red tiene un número finito de lugares pueden darse un número finito de aceleraciones que generen nuevos ω. Por eso consideramos que tenemos el segundo caso de aceleración.

Sea $S = \{x_j \mid j \geq i\}$ un conjunto infinito de nodos activos cuyos marcajes son incomparables entre sí. Esto es imposible puesto que el conjunto de todos los posibles ω-marcajes de la red es finito al ser P finito.

Definimos $S_j = \{x_k \mid j \geq k \geq i\}$ y probemos que para cada $j \geq i$, S_j contiene nodos activos cuyos marcajes son incomparables dos a dos. Consideremos un determinado S_i. Cuando se crea el nodo x_i se declara activo, ya que si no no se construyen sus sucesores. Sea $j \geq i$ y asumamos que lo anterior se cumple para S_j. Consideremos x_{j+1} y probemos que es activo y que los marcajes de S_{j+1} son incomparables. Dado que nada más crearse un nodo se declara activo, la primera parte está demostrada. Así, ningún nodo activo lo cubre. Como los elementos de S_j son activos, $\Lambda(x_{j+1}) \not\leq \Lambda(x)$ para $x \in S_j$. Además, $\Lambda(x_{j+1}) = \text{Post}(\Lambda(x_j), t_j)$, lo que implica que no se ha aplicado el primer tipo de aceleración y que, por tanto, no desactiva ningún nodo anterior. Como consecuencia, $\Lambda(x_{j+1}) \not> \Lambda(x)$ para todo $x \in S_j$. Concluimos

que los elementos de S_{j+1} son incomparables dos a dos. □

Teorema 7.2.2. *El algortimo de Poda-Monótona es coherente.*

La demostración de la completitud del algoritmo se presenta para Redes de Petri Ampliadas.

Definición 7.2.1. Una Red de Petri Ampliada (en inglés *widened Petri net*) es un par (PN, M_φ) compuesto por una Red de Petri y un marcaje $M_\varphi \geq M_0$.

Para ello introducimos las siguientes definiciones:

Definición 7.2.2. Sean una función de aceleración Acc y un árbol etiquetado $\mathbf{C} = (X, B, x_0, \Lambda)$ obtenido de la ejecución del algoritmo utilizando Acc como función de aceleración. Una función de concretización $\gamma : B^* \to T^*$ es aquella que asocia a cada camino en $\mathbf{C}$ una secuencia de transiciones de la Red de Petri

Definición 7.2.3. Consideramos que una función de aceleración Acc es coherente si admite una función de concretización γ tal que se cumple la siguiente propiedad: sean $x, y \in \mathbf{C}$, $w \in B^*$ un camino en $\mathbf{C}$ entre x e y; entonces $\forall \rho_p \preceq \gamma(w)$ existen dos nodos x' e y' tales que:

1. $\text{Post}(\Lambda(y), \rho_p) \geq \Lambda(y')$,
2. x' es un antecesor de x que se ha usado en la aceleración del nodo y_1 del camino de y a x,
3. y' está entre x' e y_1.

Lema 7.2.1. *Dada una función de aceleración Acc coherente y su concretización γ, se tiene la siguiente propiedad: dados tres nodos $x, z \in Act$ e $y \in Ancestor_C(x)$, definimos $\rho = \gamma(path_C(y, x))$. Si $\Lambda(z) \geq Post(\Lambda(y), \rho_p)$ para algún $\rho_p \preceq \rho$ entonces $y \in Ancestor_C(z)$.*

Demostración. Por la definición anterior existen nodos x', y' tales que:

- $\text{Post}(\Lambda(y), \rho_p) \geq \Lambda(y')$,
- x' es un antecesor de x que se ha usado en la aceleración del nodo y_1 del camino de y a x,
- y' está entre x' e y_1.

Asumamos, para llegar a una contradicción, que $y' \notin \text{Ancestor}_C(z)$. Asumamos también que $y_1 \notin \text{Ancestor}_C(z)$. Al aplicar el algoritmo, el nodo desactiva todo lo que hay bajo x' excepto él mismo. Como tenemos que y' está entre x' e y_1, y' es un antecesor de z, pero y_1 no lo es, y por lo tanto y_1 desactiva z, lo que es una contradicción. Tenemos, por tanto, que $y_1 \in \text{Ancestor}_C(z)$. Y esto implica $y \in \text{Ancestor}_C(z)$. □

La prueba de la corrección del algoritmo recae en el siguiente invariante, **P**, del algoritmo: para todo marcaje alcanzable M existe $(x, \rho) \in \text{Act} \times T^*$ tal que:

1. $\text{Post}(\Lambda(x), \rho_p) \geq M$
2. $\rho \neq \epsilon \Rightarrow (x, \text{first}(\rho)) \in$Wait
3. $\forall \epsilon \neq \rho_p \preceq, \nexists z \in \text{Act}$ tal que $\Lambda(z) \geq \text{Post}(\Lambda(x), \rho_p)$

Es decir, dado un marcaje alcanzable M, existe un par (x, ρ) que permite cubrirle, cuya exploración está todavía a la espera (Wait) y no se detendrá por culpa de ningún otro nodo activo.

Lema 7.2.2. *Si el algoritmo de Poda-Monótona usa una función de aceleración coherente, entonces satisface* **P** *en cada paso de su ejecución.*

Demostración. Por inducción sobre el número de pasos del algoritmo. Inicialmente (caso base), el invariante se satisface de manera trivial. Supongamos que se cumple $\mathbf{P}(k)$ y veamos que se tiene también $\mathbf{P}(k+1)$.

Sea M un marcaje alcanzable, por cumplirse $\mathbf{P}(k)$ existe (x, ρ) verificando el invariante. Consideramos tres casos dependiendo de lo que ocurra en el bucle While.

1. Si x no es desactivado ni ningún sucesor suyo con prefijo ρ está cubierto por n' podemos elegir el par (x, ρ).
2. En otro caso, asumimos que algún sucesor de x con un prefijo de ρ está cubierto por n'. Sea ρ_1 el prefijo más largo de ρ tal que $\Lambda(n') \geq \text{Post}(\Lambda(x), \rho_1)$. Supongamos que podemos elegir el par (n', ρ') donde $\rho' = \rho_1^{-1} \cdot \rho$ siendo $\rho_1^{-1} = t_1 t_2 ... t_n$ si $\rho_1 = t_n ... t_2 t_1$. Veamos si verifica el invariante:
   - La primera propiedad se sigue de la monotonía de las Redes de Petri y de $\Lambda(n') \geq \text{Post}(\Lambda(x), \rho_1)$.
   - La segunda propiedad se tiene por la línea 9 del algoritmo.

- Para probar que se cumple la tercera propiedad procedemos por reducción al absurdo. Asumamos que existen un prefijo no vacío de ρ, ρ'_p, y un nodo activo, z, tales que $\Lambda(z) \geq \text{Post}(\Lambda(n'), \rho'_p)$. Entonces $\text{Post}(\Lambda(x), \rho_1\rho'_p) \leq \text{Post}(\Lambda(n'), \rho'_p) \leq \Lambda(z)$. Como $\epsilon \neq \rho_1\rho'_p \preceq \rho$, la tercera propiedad del invariante implica que z es un nuevo nodo activo, esto es, $z = n'$. Contradicción con la elección de ρ_1.

3. En otro caso, x es desactivado por n': n' denomina un antecesor y de x tal que $y \in \text{Act}$ o $y \notin \text{Ancestor}_C(n')$. Sea w el camino en $\mathbf{C}$ entre x e y, definimos $\rho_0 = \gamma(w)$ y ρ_1 como el prefijo más largo de ρ_0 tal que $\Lambda(n') \geq \text{Post}(\Lambda(n'), \rho_1)$. Escribimos $\rho_0 = \rho_1\rho_2$ y veremos que $(n', \rho_2\rho)$ satisfacen las propiedades del invariante. Las propiedades 1 y 2 se tienen por la monotonía de las Redes de Petri y del hecho de que n' acaba de añadirse a $\mathbf{C}$. La tercera propiedad la probaremos por contradicción. Asumamos que existe ρ_p un prefijo no vacío de $\rho_2\rho$ y un nodo activo z tal que $\Lambda(z) \geq \text{Post}(\Lambda(n'), \rho_p)$. Por monotonía, $\Lambda(z) \geq \text{Post}(\Lambda(y), \rho_p)$. Si $z = n'$, ρ_p debe ser un prefijo de ρ_2, pero esto contradice la definición de ρ_2. Si $z \neq n'$, por la tercera propiedad del invariante para (x, ρ), ρ_p es un prefijo de ρ_2. Consideramos el prefijo $\rho_1\rho_p$ de ρ_0. Por el lema anterior, $y \in \text{Ancestor}_C(z)$. Como z se desactiva para construir n', llegamos a una contradicción al suponerlo activo.

□

Teorema 7.2.3. *Si la función de aceleración es coherente, entonces el algoritmo de Poda-Monótona es completo para las Redes de Petri Ampliadas.*

Demostración. Se tiene por el lema anterior y la terminación del algoritmo. Sea un marcaje alcanzable M y el conjunto Act que devuelve el algoritmo, existe un par (x, ρ) como en $\mathbf{P}$. Como Wait está vacía, tenemos $\rho = \epsilon$, por lo que el invariante nos da la completitud de Act. □

Como para toda Red de Petri PN existe un marcaje M_φ tal que $PN = WPN$, donde $WPN = (PN, M_\varphi)$ es una Red de Petri Ampliada, entonces se cumple que el algoritmo de Poda-Monótona es completo para las Redes de Petri.

Dado que el algoritmo es coherente y completo entonces es correcto.

Veamos cómo se obtiene el conjunto de mínima cobertura de la Red de Petri de la figura 3.2 por medio de este algoritmo.

Ejemplo 7.2.1. *Vamos a utilizar como función de aceleración* Acc_{all}. *Inicializamos* x_0 *el nodo con marcaje* $M_0 = (1, 0, 0)$, $X = \{x_0\}$, $Act = X$, $B = \emptyset$ *y* $Wait = \{(x_0, t_0), (x_0, t_1)\}$.

Paso 1. Eliminamos el par (x_0, t_0) *de Wait y disparamos* t_0 *desde* M_0 *obteniendo el marcaje* $M = (0, 1, 0)$. *Así tenemos un nuevo nodo* n_1 *con marcaje* $(0, 1, 0)$. *Actualizamos:*

- $X = \{x_0, n_1\}$
- $Act = \{(1, 0, 0), (0, 1, 0)\}$
- $B = \{(x_0, t_0, n_1)\}$
- $Wait = \{(x_0, t_1), (n_1, t_2)\}$

Paso 2. Partimos del marcaje (1,0,0) y disparamos t_1 *obteniendo* $M = (0, 0, 1)$. *Sea* n_2 *el nodo con este nuevo marcaje, actualizamos:*

- $X = \{x_0, n_1, n_2\}$
- $Act = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$
- $B = \{(x_0, t_0, n_1), (x_0, t_1, n_2)\}$
- $Wait = \{(n_1, t_2), (n_2, t_3)\}$

Paso 3. $(0, 1, 0) \xrightarrow{t_2} (0, 0, 2)$. *Aceleramos este marcaje obteniendo* $(0, 0, \omega)$. *Llamamos* n_3 *al nodo tal que* $\Lambda(n_3) = (0, 0, \omega)$ *y actualizamos:*

- $X = \{x_0, n_1, n_2, n_3\}$
- $Act = \{(1, 0, 0), (0, 1, 0), (0, 0, \omega)\}$. *Hemos eliminado el marcaje (0,0,1), pues el nuevo marcaje le cubre.*
- $B = \{(x_0, t_0, n_1), (x_0, t_1, n_2), (n_1, t_2, n_3)\}$
- $Wait = \{(n_2, t_3), (n_3, t_3)\}$

Paso 4. $(0, 0, 1) \xrightarrow{t_3} (0, 1, 0)$. *Como este marcaje ya está en Act solo actualizamos* $Wait = \{(n_3, t_3)\}$.

Paso 5. $(0, 0, \omega) \xrightarrow{t_3} (0, 1, \omega)$. *Sea* n_4 *el nodo asociado a este marcaje, actualizamos:*

- $X = \{x_0, n_1, n_2, n_3, n_4\}$

- $Act = \{(1,0,0),(0,1,\omega)\}$
- $B = \{(x_0,t_0,n_1),(x_0,t_1,n_2),(n1,t_2,n_3),(n_3,t_3,n_4)\}$
- $Wait = \{(n_4,t_2),(n_4,t_3)\}$

Paso 5. $(0,1,\omega) \xrightarrow{t_2} (0,0,\omega)$. *Como este marcaje está cubierto por el de partida solo actualizamos* $Wait = \{(n_4,t_3)\}$.

Paso 6. $(0,1,\omega) \xrightarrow{t_3} (0,2,\omega)$. *Aceleramos este marcaje obteniendo* $(0,\omega,\omega)$ *y denominamos por* n_5 *al nodo tal que* $\Lambda(n_5) = (0,\omega,\omega)$. *Actualizamos:*

- $X = \{x_0,n_1,n_2,n_3,n_4,n_5\}$
- $Act = \{(1,0,0),(0,\omega,\omega)\}$
- $B = \{(x_0,t_0,n_1),(x_0,t_1,n_2),(n1,t_2,n_3),(n_3,t_3,n_4),(n_4,t_3,n_5)\}$
- $Wait = \{(n_5,t_2),(n_5,t_3)\}$

Paso 7. Como desde el marcaje (0,ω,ω) disparando tanto t_2 *como* t_3 *obtenemos el mismo marcaje que el de partida actualizamos* $Wait = \emptyset$ *y termina el algoritmo. Así el conjunto de mínima cobertura es* $\{(1,0,0),(0,\omega,\omega)\}$ *y el árbol* $\mathbf{C} = (X,B,x_0,\Lambda)$ *es el siguiente:*

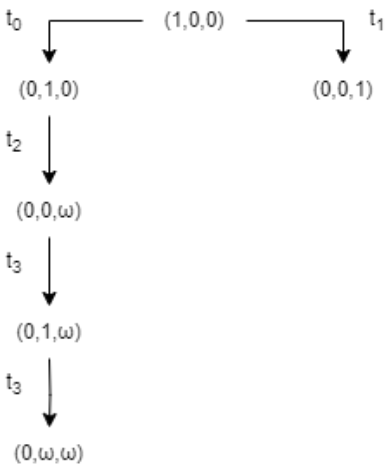

Figura 7.2

Capítulo 8

Algoritmo MinCov

8.1. Motivación

La idea principal es tomar como punto de partida el algoritmo de Karp-Miller y aplicarle distintas modificaciones para desarrollar un algoritmo con menor tiempo de ejecución que los anteriores. Introduciremos el concepto de abstracción como una ω-transición que imita el efecto de un conjunto infinito de secuencias de disparo. Así, añadir abstracciones a una Red de Petri no modifica su conjunto de cobertura y la aceleración de Karp-Miller (figura 5.1) puede formalizarse como una abstracción cuya incidencia en los lugares es ω o nula. Se ha encontrado una forma de solucionar el error del algoritmo del árbol de mínima cobertura: memorizar las aceleraciones descubiertas y usarlas como si fueran transiciones.

A continuación se presentará un nuevo algoritmo basado en estas ideas que requiere menos memoria que los anteriores y su tiempo de ejecución es muy rápido.

8.2. Abstracciones de cobertura

Van a tener gran importancia la matriz de incidencia previa, $\mathbf{Pre} \in \mathbb{N}^{P\times T}$, y la matriz de incidencia, $\mathbf{C} \in \mathbb{Z}^{P\times T}$, de una Red de Petri.

Definición 8.2.1 (Matriz de incidencia)**.** Sea $PN = (P, T, W, M_0)$ una Red de Petri. La matriz de incidencia $\mathbf{C}$ está dada por:

$$\mathbf{C}(t)(p) = W(t,p) - W(p,t)$$

La matriz de incidencia previa, **Pre**, se define como

$$\mathbf{Pre}(t)(p) = W(p,t)$$

El vector columna de cada una de estas matrices indexado por $t \in T$ se denota por $\mathbf{Pre}(t)$ o $\mathbf{C}(t)$. Desde un marcaje M se puede disparar una transición t si $M \geq \mathbf{Pre}(t)$, y el marcaje resultante es $M' = M + \mathbf{C}(t)$.

Para introducir abstracciones y aceleraciones antes tenemos que generalizar las transiciones para que puedan marcar un lugar con ω tokens. Para ello introducimos la siguiente definición:

Definición 8.2.2 (ω-transición)**.** Dado un conjunto de lugares P, una ω-transición a está definida por $\mathbf{Pre}(a) \in \mathbb{Z}_\omega^P$ y $\mathbf{C}(a) \in \mathbb{Z}_\omega^P$ tal que $\mathbf{Pre}(a) + \mathbf{C}(a) \geq 0$ y $\mathbf{Pre}(a)(p) = \omega$ implica $\mathbf{C}(a)(p) = \omega$.

Sean a y a' dos ω-transiciones, decimos que $a \leq a'$ si $\mathbf{Pre}(a) \leq \mathbf{Pre}(a') \wedge \mathbf{C}(a) \geq \mathbf{C}(a')$.

Sea $\sigma = t\sigma'$ con t una ω-transición y σ' una secuencia de ω-transiciones, entonces $\mathbf{C}(\sigma) = \mathbf{C}(t) + \mathbf{C}(\sigma')$, y para todo lugar p si $\mathbf{C}(t)(p) = \omega$ se tiene que $\mathbf{Pre}(\sigma)(p) = \mathbf{Pre}(t)(p)$ y, en caso contrario, $\mathbf{Pre}(\sigma)(p) = \text{máx}(\mathbf{Pre}(t)(p), \mathbf{Pre}(\sigma')(p) - \mathbf{C}(t)(p))$.

Definición 8.2.3. Dada una red de Petri, sus matrices $\mathbf{Pre}$ y $\mathbf{C}$, y una ω-transición a, a es una abstracción si $\forall n \geq 0$ existe $\sigma_n \in T^*$tal que para todo lugar p con $\mathbf{Pre}(a)(p) \in \mathbb{N}$ se tiene:

- $\mathbf{Pre}(\sigma_n)(p) \leq \mathbf{Pre}(a)(t)$.
- Si $\mathbf{C}(a)(p) \in \mathbb{Z}$ entonces $\mathbf{C}(\sigma_n)(p) \geq \mathbf{C}(a)(p)$.
- Si $\mathbf{C}(a)(p) = \omega$ entonces $\mathbf{C}(\sigma_n)(p) \geq n$.

La siguiente proposición justifica el interés que se tiene por las abstracciones:

Proposición 8.2.1. *Sean PN una Red de Petri, $Cover(PN)$ su conjunto de cobertura, a una abstracción y M un ω-marcaje tal que $[\![M]\!] \subseteq Cover(PN)$ y $M \xrightarrow{a} M'$, entonces $[\![M']\!] \subseteq Cover(PN)$.*

Demostración. Por ser a una abstracción, dado n existe $\sigma_n \in T^*$ como en la definición anterior. Tomemos $M^* \in [\![M']\!]$ y denotemos $N = \text{máx}\{M^*(p) | M'(p) = \omega\}$ y $L = \text{máx}\{\mathbf{Pre}(\sigma_n)(p), N - \mathbf{C}(\sigma_n)(p) | M'(p) = \omega\}$.

Definamos $M^\sharp \in [\![M]\!]$ para cada $p \in P$ como $M^\sharp(p) = M(p)$ si $M(p) < \omega$ y $M^\sharp(p) = L$ en otro caso. Veamos que σ_n puede ejecutarse en este marcaje. Sea $p \in P$, si $M(p) < \omega$ entonces $M^\sharp(p) = M(p) \geq \mathbf{Pre}(a)(p) \geq \mathbf{Pre}(\sigma_n)(p)$

y, en caso contrario, $M^\sharp(p) = L \geq \mathbf{Pre}(\sigma_n)(p)$.

Veamos que $M^\sharp + \mathbf{C}(\sigma_n) \geq M^*$. Sea $p \in P$, diferenciamos tres casos:

- Si $M(p) < \omega$ y $\mathbf{C}(a)(p) < \omega$ entonces $M^\sharp(p) + \mathbf{C}(\sigma_n)(p) \geq M(p) + \mathbf{C}(a)(p) = M'(p) \geq M^*(p)$.
- Si $M(p) < \omega$ y $\mathbf{C}(a)(p) = \omega$ entonces $M^\sharp(p) + \mathbf{C}(\sigma_n)(p) \geq \mathbf{C}(\sigma_n)(p) \geq N \geq M^*(p)$.
- Si $M(p) = \omega$ entonces $M^\sharp(p) + \mathbf{C}(\sigma_n)(p) \geq N - \mathbf{C}(\sigma_n)(p) + \mathbf{C}(\sigma_n)(p) = N \geq M^*(p)$.

Así, $M^\sharp + \mathbf{C}(\sigma_n) \in [\![M']\!]$. Pero como $M^\sharp \in [\![M]\!] \subseteq \text{Cover}(PN)$ entonces $M^\sharp + \mathbf{C}(\sigma_n) \in \text{Cover}(PN)$. Concluimos que $[\![M']\!] \subseteq Cover(PN)$. □

Una forma sencilla de construir abstracciones es concatenándolas: sea σ una secuencia de abstracciones entonces la ω-transición a definida por $\mathbf{Pre}(a) = \mathbf{Pre}(\sigma)$ y $\mathbf{C}(a) = \mathbf{C}(\sigma)$, es una abstracción.

Definición 8.2.4. Sea a una abstracción, si definimos la ω-transición a' de la siguiente manera para todo $p \in P$:

- Si $\mathbf{C}(a)(p) < 0$ entonces $\mathbf{Pre}(a')(p) = \mathbf{C}(a')(p) = \omega$.
- Si $\mathbf{C}(a)(p) = 0$ entonces $\mathbf{Pre}(a')(p) = \mathbf{Pre}(a)(p)$ y $\mathbf{C}(a')(p) = 0$.
- Si $\mathbf{C}(a)(p) > 0$ entonces $\mathbf{Pre}(a')(p) = \mathbf{Pre}(a)(p)$ y $\mathbf{C}(a')(p) = \omega$.

Entones a' es una aceleración.

Dadas una Red de Petri y una aceleración a, la ω-transición $trunc(a)$ definida como:

- $\mathbf{C}(trunc(a)) = \mathbf{C}(a)$.
- $\forall p \in P$ tal que $\mathbf{Pre}(a)(p) \neq \omega$, $\mathbf{Pre}(trunc(a))(p) = \text{mín}(\mathbf{Pre}(a)(p), e(3de)^{(d+1)^{2d+4}})$, donde $d = |P|$ y $e = \text{máx}_{p,t}(\mathbf{Pre}(t)(p), \mathbf{Pre}(t)(p) + \mathbf{C}(t)(p))$.
- $\forall p \in P$ tal que $\mathbf{Pre}(a)(p) = \omega$ entonces $\mathbf{Pre}(trunc(a))(p) = \omega$.

es una aceleración. Usando esta noción se van a construir las aceleraciones que usa el algoritmo.

8.3. Algoritmo

Este algoritmo [5], al contrario que los anteriores, va a memorizar las aceleraciones en lugar de los ω-marcajes. Fue propuesto por Haddad, Finkel y Khmelnitsky en 2020, y es el algoritmo más nuevo y diferente para computar el conjunto de mínima cobertura de una red.

El algoritmo 8.1 construye un árbol etiquetado $CT = (N, B, \delta, \Lambda)$ y un conjunto Acc de ω-transiciones (que son, además, aceleraciones) a partir de una Red de Petri. Cada $v \in N$ está etiquetado con un ω-marcaje, $\Lambda(v) \in \mathbb{N}_\omega^P$, y su predecesor se denota por $prd(v)$. Cada $e \in B$ está etiquetada por una secuencia de disparo $\delta(e) \in T \cdot \text{Acc}^*$ que consiste en una transición ordinaria seguida de una secuencia de aceleraciones.

En el bucle se toma un nodo $u \in$Front. Desde $\Lambda(u)$ se dispara una secuencia σ alcanzando un marcaje M_u que maximiza la cantidad de ω obtenidos. El algoritmo actualiza $\lambda(u)$ con M_u y la etiqueta del vértice entrante a u concatenando σ. A partir de aquí y dependiendo de $\Lambda(u)$ se toma uno de los siguientes tres caminos:

- Limpieza. Si $\exists u' \in N \setminus$ Front tal que $\Lambda(u') \geq \Lambda(u)$, entonces u es redundante y se elimina.
- Aceleración. Si existe un antecesor u' de u tal que $\Lambda(u') < \Lambda(u)$, se puede computar una aceleración a. Se añade u' a Front.
- Exploración. En otro caso, se poda y elimina todo u' tal que $u' \in N$ y $\Lambda(u') < \Lambda(u)$, pues son redundantes. Después, se elimina u de Front y para toda transición ejecutable desde $\Lambda(u)$, se crea un nuevo nodo que se va a añadir a Front.

8.3.1. Corrección del algoritmo

Para probar la corrección del algoritmo hay que demostrar que termina, que los ω-marcajes asociados a los nodos del árbol son incomparables (es decir, $\Lambda(N)$ es una anticadena), su consistencia ($[\![\Lambda(N)]\!] \subseteq Cover(PN)$) y su completitud ($Cover(PN) \subseteq [\![\Lambda(N)]\!]$).

Proposición 8.3.1. *El algoritmo MinCov termina.*

Demostración. Vamos a razonar con una variante teórica del algoritmo que en lugar de borrar un nodo vía Delete(u') lo marca como 'podado' extrayéndolo de Front. Además, en lugar de cortar un subárbol cuando el marcaje del

```
1  procedure MinCov(PN, M_0) return CT:
2  N = {r}; B = ∅; Front= {r}; Λ(r) = M_0; Acc= ∅; δ(r,r) = ε
3  while Front≠ ∅ do
4    select u ∈Front
5    σ ∈ Acc* maximal fireable sequence of accelerations
       from Λ(u)
6    Λ(u) = Λ(u) + C(σ)
7    δ((prd(u), u)) = δ((prd(u), u)) · σ
8    if ∃u' ∈ N \ Front s.t. Λ(u') ≥ Λ(u) then Delete(u)
9    if ∃u' ∈ Anc(N) s.t. Λ(u) > Λ(u') then
10     γ ∈ B* path from u' to u in CT
11     a =NewAcceleration
12     for each p ∈ P do
13       if C(δ(γ))(p) < 0: Pre(a)(p) = ω; C(a)(p) = ω
14       if C(δ(γ))(p) = 0: Pre(a)(p) = Pre(δ(γ))(p); C(a)(p) = 0
15       if C(δ(γ))(p) > 0: Pre(a)(p) = Pre(δ(γ))(p); C(a)(p) = ω
16     a = trunc(a); Acc=Acc∪{a}; Prune(u');
17     Front=Front∪{u'}
18   else
19     for u' ∈ N do
20       if Λ(u') < Λ(u) then Prune(u'); Delete(u')
21     Front=Front\{u}
22     for each t ∈ T ∧ Λ(u) ≥ Pre(t) do
23       u' =NewNode; N = N ∪ {u'}; Front=Front∪{u'};
24       B = B ∪ {(u, u')}
25       Λ(u') = Λ(u) + C(t); δ((u, u')) = t
```

Figura 8.1: Algoritmo MinCov

actual nodo sea mayor que el de un nodo que no sea antecesor del actual, los marcamos como 'podados' y los extraemos de Front. Además, en el caso de que el marcaje del nodo actual sea mayor que el de su antecesor, en lugar de cortar el subárbol, marcamos el camino del antecesor al actual como 'acelerado' y los nodos del subárbol como 'podados' y añadimos el nodo actual a Front con el marcaje de su nodo antecesor. Todos los marcajes del subárbol se extraen de Front.

Ignoramos todos los nodos marcados como 'podados' y 'acelerados'. Este nuevo algoritmo funcionaría como MinCov pero con la diferencia de que el tamaño del árbol no decrecería nunca. Con lo cual, el algoritmo no termi-

naría y construyéndose un árbol infinito. Por ello, contiene un camino infinito donde ninguno de sus nodos puede ser marcado como 'podado', pues pertenecerían a un subárbol finito.

Observese que el marcaje del nodo que sigue un subcamino acelerado tiene al menos un ω más que el marcaje del primer nodo de ese subcamino. Por tanto, existiría un subcamino infinito con nodos no marcados en N. Pero $\mathbb{N}_\omega^P$ está bien ordenado, así que existirán dos nodos v y v', descendiente de v, tales que $\Lambda(v') \geq \Lambda(v)$, contradiciendo el comportamiento del algoritmo. □

Para probar que $\Lambda(N)$ es una anticadena vamos a introducir la siguiente notación: nos referiremos por $CT_n = (N_n, B_n, \delta_n, \Lambda_n)$, Front_n y Acc_n a los valores de CN, Front y Acc en la iteración n del algoritmo.

Proposición 8.3.2. *Para todo $n \in \mathbb{N}$, $\Lambda(N_n \setminus \mathit{Front}_n)$ es una anticadena. Luego, por terminación, $\Lambda(N)$ es una anticadena.*

Demostración. Por comodidad introducimos la notación $N' := N \setminus \text{Front}$ y $N'_n := N_n \setminus \text{Front}_n$. Probaremos por inducción sobre n que N'_n es una anticadena.

Inicialmente $N_0 = \text{Front}_0 = \{r\}$, así que $N'_0 = \emptyset$, que es una anticadena.

Supongamos que N'_n es una anticadena. Podemos modificarla añadiendo o eliminando nodos de N_n y eliminando nodos de Front_n manteniéndolos en N_n. Para ello el algoritmo nos ofrece varias alternativas:

- Llamar a las funciones Delete y Prune. Estas no añade nuevos nodos a N'_n, así que se preserva la anticadena.
- En la línea 23 se puede añadir simultáneamente un nodo a N y Front, así que la anticadena queda intacta.
- Si solo añadimos nodos a Front, los eliminamos de N'_n, y la anticadena se sigue preservando.
- Solo podemos añadir un nodo a N' eliminándolo de Front pero manteniéndolo en N. Esto ocurre en la línea 21 del algoritmo con el nodo u. Entonces se tiene que todos los marcajes de $\Lambda(N'_n) \subseteq \Lambda(N_n)$ son menores o no son comparables con $\Lambda_{n+1}(u)$, es decir, no se cumplen la condiciones de las líneas 8 y 9. En las líneas 18-20, se eliminan todos los nodos $n' \in N'_n \subseteq N_n$ tales que $\Lambda_{n+1}(u') < \Lambda_{n+1}(u)$. Denotemos por $N''_n \subseteq N'_n$ al conjunto N' al final de la línea 20. Entonces

$\Lambda_{n+1}(u)$ es incomparable con los marcajes de $\Lambda_{n+1}(N''_n)$. Esto junto a que por $N''_n \subseteq N'_n$, $\Lambda_{n+1}(N''_n)$ es una anticadena, concluimos que $\Lambda_{n+1}(N'_{n+1}) = \Lambda_{n+1}(N''_n) \cup \{\Lambda_{n+1}(u)\}$ es una anticadena.

□

Los siguientes lemas, de los cuales no daremos su demostración, se necesitan para la demostración de la proposición 8.3.3. que se encuentra a continuación:

Lema 8.3.1. *Para todo $n \in \mathbb{N}$, para todo $u \in N_n \setminus \{r\}$, $\Lambda_n(prd(u)) \xrightarrow{\delta(prd(u),u)} \Lambda_n(u)$ y $M_0 \xrightarrow{\delta(r,r)} \Lambda_n(r)$.*

Lema 8.3.2. *Acc es un conjunto de aceleraciones independientemente del punto en el que nos encontremos en la ejecución del algoritmo MinCov.*

Proposición 8.3.3. $[\![\Lambda(N)]\!] \subseteq Cover(PN)$

Demostración. Sea $v \in N$ y consideremos el camino $u_0 = r, u_1, ..., u_k = v$ de CT desde la raíz hasta v. Denotemos por $\sigma \in (T \cup \mathrm{Acc})^*$ a $\delta(prd(u_0), u_0) \cdots \delta(prd(u_k), u_k)$. Por el primero de los lemas anteriores, $M_0 \xrightarrow{\sigma} \Lambda(v)$, y por el segundo, σ es una secuencia de abstracciones. Como la ω-transición a definida como $\mathbf{Pre}(a) = \mathbf{Pre}(\sigma)$ y $\mathbf{C}(a) = \mathbf{C}(\sigma)$ es una abstracción por la proposición 8.2.1., $[\![\Lambda(v)]\!] \subseteq Cover(PN)$. □

A continuación se presentan dos definiciones y una proposición que serán indispensables para probar la completitud del algoritmo:

Definición 8.3.1. La secuencia de disparo $M \xrightarrow{\sigma} M'$, donde $\sigma = \sigma_0 t_1 \sigma_1 ... t_k \sigma_k$ con $t_i \in T$ y $\sigma_i \in \mathrm{Acc}^*$ para todo i, es una secuencia de exploración si existe $v \in$Front con $\Lambda(v) = M$ y si para todo $0 \leq i \leq k$ no existe $v' \in V \setminus$Front con $M + \mathbf{C}(\sigma_0 t_1 \sigma_1 ... t_i \sigma_i) \leq \Lambda(v')$.

Definición 8.3.2. Un marcaje M está cuasi-cubierto si o bien existe $v \in N \setminus$Front tal que $\Lambda(v) \geq M$, o bien existe una secuencia de exploración $M' \xrightarrow{\sigma} M'' \geq M$.

Proposición 8.3.4. *Al comienzo de cada iteración, todo $M \in Cover(PN)$ está cuasi-cubierto.*

Con esta proposición se tiene inmediatamente el siguiente resultado:

Proposición 8.3.5. *Cuando MinCov termina, se tiene $Cover(PN) \subseteq [\![\Lambda(N)]\!]$.*

Pues por la proposición anterior, todo marcaje de $Cover(PN)$ está cuasi-cubierto. Como el algoritmo termina, Front es vacío para todo $M \in Cover(PN)$, entonces existe $v \in N$ tal que $M \leq \Lambda(v)$.

Vamos a presentar brevemente cómo actuaría este algoritmo para la Red de Petri de la figura 3.2:

Ejemplo 8.3.1. *Partimos del marcaje inicial* $M_0 = (1, 0, 0)$. *Ejecutando la cadena de transiciones* t_0, t_2 *obtenemos* $(1, 0, 0) \xrightarrow{t_0} (0, 1, 0) \xrightarrow{t_2} (0, 0, 2)$. *Tenemos hasta este momento un árbol compuesto por estos tres marcajes. Disparamos ahora la transición* t_3*:* $(1, 0, 0) \xrightarrow{t_0} (0, 1, 0) \xrightarrow{t_2} (0, 0, 2) \xrightarrow{t_3} (0, 1, 1)$. *Tenemos que* $(0, 1, 1) \geq (0, 1, 0)$*, con lo cual, tenemos una aceleración* $a = t_2 t_3$ *que añade* ω *a* p_1. *Podamos el árbol anterior quedándonos con los marcajes* $(1, 0, 0)$ *y* $(0, 1, \omega)$

Continuamos, $(0, 1, \omega) \xrightarrow{t_2} (0, 0, \omega)$*, que está cubierto por el marcaje anterior.*

Ejecutamos la transición t_3*,* $(0, 1, \omega) \xrightarrow{t_3} (0, 2, \omega)$*, y tenemos la aceleración* $a = t_3$. *Podamos el árbol, que tendría los marcajes* $(1, 0, 0)$ *y* $(0, \omega, \omega)$. *Estos dos constituyen el conjunto de mínima cobertura de la Red de Petri.*

Capítulo 9

Conclusiones

A continuación discutiremos la eficiencia de los anteriores algoritmos teniendo en cuenta los resultados de los estudios realizados en los distintos artículos de la bibliografía.

Aunque el algoritmo de Finkel no proporciona el conjunto de mínima cobertura de una Red de Petri, sí que nos da una aproximación de este y en muchos casos este algoritmo computa correctamente el árbol de mínima cobertura. Con lo cual, supone un buen punto de partida para la búsqueda de algoritmos que computen el conjunto de mínima cobertura.

Se han llevado a cabo comparaciones entre este algoritmo y el de Karp-Miller que prueban que el algoritmo de Finkel es bastante más veloz y genera un árbol mucho más pequeño. Esto último no nos extraña pues el procedimiento de Karp-Miller da un sobrecubrimiento del conjunto de mínima cobertura.

Para probar la eficiencia del algoritmo CovProc y de la importancia del oráculo, se han llevado a cabo comparaciones entre los algoritmo de Karp-Miller y CovProc con y sin oráculo (mismo algoritmo pero el oráculo siempre es el conjunto vacío). En estos casos los conjuntos de pares construidos por CovProc son pequeños en relación con el tamaño del árbol de Karp-Miller, aunque el número de pares que se han ido creando a lo largo de la ejecución del mismo algoritmo no es tan pequeño en relación con el árbol de Karp-Miller.

Se han llevado a cabo dos tipos de pruebas de rendimiento en [5] para comparar los algoritmos MinCov y CovProc. Estas han consistido en 123 pruebas de rendimiento estándar y 100 Redes de Petri generadas aleatoriamente pero con las siguientes características: $50 < |P|, |T| < 100$ y el número de lugares

conectados a cada transición es como máximo 10. En la primera prueba se ha obtenido que el número de veces que se agotó el tiempo de espera fue de 16 en MinCov y 49 en CovProc; y el tiempo total de las instancias que no han agotado el tiempo de espera en más de 900 segundos es de 18.127 en el caso de MinCov y de 47.081 en CovProc. En cuanto a la segunda prueba, el número de veces que se ha agotado el tiempo de espera en MinCov es de 14 y en CovProc, 80; y el tiempo total cuando no se ha alcanzado el tiempo límite es de 13.989 en el caso de MinCov y para CovProc, 74.767. Con lo que se concluye que MinCov es mucho más rápido que CovProc.

CovProc destaca por utilizar conjuntos de pares de marcajes y su estrategia de minimizar dichos conjuntos. Aunque es realmente útil, en comparación con los demás algoritmos, tiene un coste en memoria muy elevado.

Por otro lado, el algoritmo StackProc puede manejar una gran cantidad de casos de estudio. Se vuelven a utilizar conjuntos de marcajes en lugar de pares de marcajes. La principal característica de este procedimiento es que utiliza una pila para almacenar los diferentes casos de estudio.

Con el algoritmo de Poda-Monótona se vuelve un poco a los orígenes, donde vamos computando el árbol y el conjunto de mínima cobertura en lugar de solo esto último. Este procedimiento se enfoca en la poda de conjuntos monótonos de marcajes, lo que mejora la eficiencia del algoritmo. Además, almacena los siguientes casos de estudio en un conjunto de nodos y transiciones. Aún así, su eficacia está ligada a la monotonía de la red, por eso si se aplica este algoritmo a una red con un comportamiento dinámico y con poca monotonía, su rendimiento se verá considerablemente reducido. A pesar de ser un algoritmo sencillo y muy fácil de utilizar, si la Red de Petri no tiene un comportamiento muy monótono, es decir, tiene un comportamiento muy dinámico, no es el algoritmo más eficiente para computar el conjunto de mínima cobertura de dicha red.

En el algoritmo MinCov se utilizan las matrices de incidencia e incidencia previa, y la gran novedad que presenta es que almacena las aceleraciones en lugar de los marcajes. Su procedimiento es bastante complejo por lo que para redes pequeñas quizá no sea el algoritmo más apropiado. Los estudios realizados en [5] han demostrado que es un algoritmo realmente rápido.

Capítulo 10

Conclusions

In the following we will discuss the efficiency of the presented algorihms taking into account the results of the studies carried out in the diverse articles of the bibliography.

Although Finkel's algorithm does not provide the minimal coverability set of a Petri Net, it does give an approximation of it and in many cases this algorithm correctly computes the minimal coverability tree. So, it is a good starting point for the search of algorithms that computes the minimal coverability set correctly.

Comparisons between this algorithm and the Karp-Miller algorithm have been carried out and prove that the Finkel algorithm is significantly faster and generates a much smaller tree. Which is not surprising as the Karp-Miller procedure gives an overcoverage of the minimal coverability set.

To test the efficiently of the CovProc algorithm and the importance of the oracle, comparisons have been made between the Karp-Miller algorithm and CovProc algorithm with and without an oracle (same algorithm but with an empty set as thd oracle). In these cases the set of pairs built by CovProc are small relative to the size of the Karp-Miller tree, although the number of pairs created during the execution of the algorithm is not so small relative to the Karp-Miller tree.

Two types of performance tests have been carried out in [5] to compare the MinCov and the CovProc algorithms. These have consisted of 123 standard performance tests and 100 randomly generated Petri Nets with the following characteristics: $50 < |P|, |T| < 100$ and the number of places connected to each transition is at most 10. In the first test, the number of times that ti-

meout was 16 in MinCov and 49 in CovProc; and the total time of instances that have not timed out more than 900 seconds is 18,127 in the case of MinCov and 47,081 in CovProc. For the second test, the number of times the timeout has been reached in MinCov is 14 and in CovProc, 80; and the total time when the timeout has not been reached is 13,989 in the case of MinCov and for CovProc, 74,767. This concludes that MinCov is much faster than CovProc.

CovProc is notable for its use of sets of pairs of markers and its strategy of minimising these sets. Although it is really useful, compared to the other algorithms, it has a very high cost in memory.

On the other hand, the StackProc algorithm can handle a large number of case studies. It again uses sets of markers instead of pairs of markers. The main feature of this procedure is that it uses a stack to store the different study cases.

With the Monotone-Prune algorithm, we go back a bit to the origins, where we compute the tree and the minimal coverability set instead of only the set. This procedure focuses on pruning monotone sets of markings, which improves the efficiency of the algorithm. Moreover, it stores the following case studies in a set of nodes and transitions. Even so, its efficiency is linked to the monotonicity of the network, so if this algorithm is applied to a network with dynamic behaviour and with little monotonicity, its performance will be considerably reduced. Despite of being a simple algorithm and easy to use, if the Petri net does not have a very monotonic behaviour, that is to say, it has a very dynamic one, it is not the most efficient algorithm to compute the minimum coverage set of the Petri net.

The MinCov algorithm uses the incidence matrix and the backward incidence matrix, and the great novelty it presents is that it stores the accelerations instead of the markings. Its procedure is quite complex, so for small networks it may not be the most appropriate algorithm. The studies carried out in [5] have shown that it is a really fast algorithm.

Bibliografía

[1] A. Finkel. *The Minimal Coverability Graph for Petri Nets*. Conference Paper, 1991.

[2] J.-F. Raskin G. Geeraerts y L. Van Begin. *On the efficient computation of the minimal coverability set for Petri nets*. Computer Science Department, Université Libre de Bruxelles (U.L.B.), 2007.

[3] A. Piipponen y A. Valmari. *Constructing Minimal Coverability Sets*. Department of Mathematics, Tampere University of Technology, 2012.

[4] P.-A. Reynier y F. Servais. *On the Computation of the Minimal Coverability Set of Petri Nets*. Aix-Marseille Univ, Université de Toulon, France y École Supérieure d'Informatique de Bruxelles, Belgium, 2019.

[5] S. Haddad A. Finkel e I. Khmelnitsky. *Minimal Coverability Tree Construction Made Complete and Efficient*. Université Paris-Saclay, Inria e Institut Universitaire de France, France, 2020.

[6] J.-F. Raskin A. Finkel G. Geeraerts y L. Van Begin. *A counter-example to the minimal coverability tree algorithm*. Article, 2005.

[7] C. Rackoff. *The covering and boundedness problems for vector addition systems*. Theor. Comput. Sci., 1978.

[8] P.-A. Reynier y F. Servais. *Minimal Coverability Set for Petri Nets: Karp and Miller Algorithm with Pruning*. LIF, Université Aix-Marseille y CNRS, France, 2013.