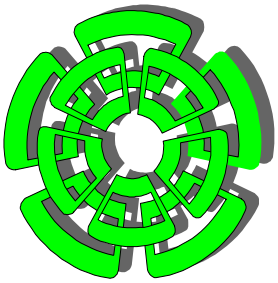

CENTRO de INVESTIGACIÓN
y de ESTUDIOS AVANZADOS
del IPN

DEPARTAMENTO DE INGENIERIA ELECTRICA
SECCION DE COMPUTACION

Detección de Candados Mortales en Base de Datos utilizando Redes de Petri

Tesis que Presenta el:

Ing. José de Jesús Trujillo Ferrara

para obtener el grado de Maestro en Ciencias en la especialidad de Ingeniería Eléctrica Opción Computación

Director de la tesis

Dra. Xiaoou Li Zhang

México, D.F. Mayo de 2004

Índice general

Agradecimientos

Al CONACYT por apoyarme con la beca de maestría y al CINVESTAV en general.

Resumen

Imaginemos que tenemos un grupo de transacciones tal que cada transacción está esperando que otra transacción del mismo grupo libere un recurso compartido. A esta situación se le denomina candado mortal; este problema se puede tratar de diferentes formas: la detección, el análisis y la recuperación [6].

Para la detección se trabajará con redes de Petri para modelar y describir gráficamente el comportamiento de un grupo de "n"transacciones que estén compitiendo por un recurso compartido en la base de datos. El grafico generado se analizará para determinar si algunas transacciones se encuentran esperando a que se libere un recurso o, a que una transacción finalice.

Si se detecta un candado mortal, se tendrá que brindar los recursos que necesita a una transacción para que termine su ejecución, dejando bloqueadas las demás transacciones; al terminar la primera transacción se debe poner en ejecución a las restantes.

En la fase de recuperación se tiene que estar en constante revisión el comportamiento general de las transacciones y los recursos de la base de datos; con la finalidad de generar otro grafico y realizar el análisis de detección, en este caso se tendrá que bloquear la ejecución de una y liberar los recursos que esté utilizando.

Para esta tesis se trabajo con un conjunto de transacciones que se deseen poner en ejecución concurrentemente. Con la finalidad de evitar candados mortales. Si se detecta que este conjunto de transacciones tienen conflictos se optará por reiniciar una transacción.

Abstract

In this thesis, we focus on database transaction deadlock detection. Transaction dependencies are modeled with Petri nets, then deadlocks of transactions are detected based on the Petri net model. The main results are:

1. Analyze dependency relation between database transactions.

2. Model transaction dependencies with Petri nets. A set of transactions can be converted into a Petri net model. A conversion algorithm is developed.

3. Use the incidence matrix techniques of Petri net model to detect deadlock. A deadlock detection algorithm is developed.

4. Implementation of TRANSimul (TRANSaction Simulation).

   4.1 TRANSimul interface design and implementation

   4.2 implementation of the conversion algorithm

   4.3 implementation of the deadlock detection algorithm

   4.4 Experiments on TRANSimul with different examples

Capítulo 1

Introducción

Una colección de varias operaciones (consultas, actualizaciones, etc.) en una base de datos son llamadas transacciones. La base de datos debe asegurar que los datos sean íntegros y que las transacciones se ejecuten satisfactoriamente. Para esto las transacciones deben tener las siguientes propiedades: [2]

- **Atomicidad**: Se deben ejecutar todas las operaciones de la transacción o ninguna.
- **Consistencia**: La ejecución de una instrucción es aislada y preserva la consistencia en la base de datos.
- **Aislamiento**: Varias transacciones se pueden ejecutar concurrentemente; el sistema debe garantizar que todas se ejecuten satisfactoriamente.
- **Durabilidad**: Después que una transacción termine satisfactoriamente, los cambios hechos por ésta deben persistir en la base de datos.

La transacción en una base de datos está compuesta por dos operaciones:

- *leer(x)*, se transfiere el dato x de la base de datos a un almacenamiento local (buffer) para la operación leer.
- *escribir(x)*, se transfiere el dato x del buffer local para escribirlos en la base de datos.

Usualmente los sistemas de base de datos permiten múltiples transacciones, es decir, que se pueden ejecutar concurrentemente. Esto nos brinda mayor eficiencia pero, si no se controla adecuadamente, puede causar problemas en la consistencia de los datos. Cuando un grupo de transacciones se van a ejecutar, es necesario generar un esquema, que contenga las operaciones, los recursos y el estado de las transacciones y, computacionalmente, este grupo de transacciones debe generar el mismo resultado que si cada transacción se ejecutara aisladamente y en forma serial; de esta forma, se garantizan las propiedades antes mencionadas.

Una forma de obligar a que una transacción se ejecute serialmente es, exigiendo que el acceso a los datos se haga de manera mutuamente excluyente. Es decir, cuando una transacción se encuentra accesando un dato, ninguna otra transacción lo puede acceder. Para esto se coloca un candado en los recursos compartidos y cuando se desea acceder al recurso se hace un requerimiento del candado. Si está disponible, se le asigna y sólo esta transacción puede "entrar" a modificar el recurso; en caso de no estar disponible la transacción tendrá que esperar a que se libere el candado. Esta manera de garantizar la serialización y, por lo tanto, el aislamiento. Pero tiene como desventaja el posible resultado de un candado mortal[8].

Imaginemos que tenemos un grupo de transacciones tal que cada transacción está esperando que otra transacción del mismo grupo libere un recurso compartido. A esta situación se le denomina candado mortal; este problema se puede tratar de diferentes formas: la detección, el análisis y la recuperación [6].

Por ejemplo, supongamos que las transacciones Ti y Tj son tareas dentro de la misma transacción y tenemos una dependencia especificada en que, el commit (comando que termina la transacción y finaliza los cambios hechos por ésta) de Ti debe preceder al de Tj (esta dependencia sólo indica el orden del commit de las dos transacciones) y se pueden ejecutar en paralelo. Figuremos que Tj adquiere el candado de escritura en el dato "x "y después Ti requiere adquirir el candado de lectura de "x". Obviamente Tj no puede liberar el candado hasta que realiza el commit; sin embargo, Tj debe esperar al commit de Ti. Ti, por otro lado, está esperando que Tj libere el candado en "x". A esta situación se le denomina candado mortal y es el que trataremos en esta tesis [1].

En la bibliografía existen dos técnicas utilizando graficos dirigido para modelar situaciones de candados mortales: Graficos de asignación de recursos (del ingles Resource-Allocation Graph RAG) y Graficos de espera (del ingles Wait For Graph WFG).

En RAG existen dos tipos de nodos:

- Rectángulos que representan procesos o transacciones (ver Figura 1.1)

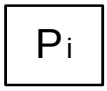

Figura 1.1: Rectángulo que representa transacción o proceso en RAG

- Círculos que representan recursos (ver figura 1.2)

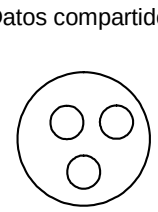

Figura 1.2: Círculo que representa recurso en RAG

Y dos tipos de arcos dirigidos:

- Arcos de Petición. P_i pide un recurso de tipo R_1. (ver figura 1.3)

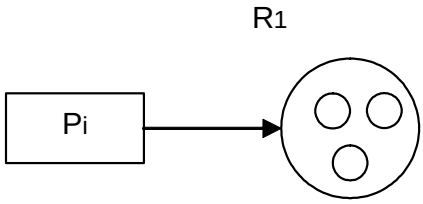

Figura 1.3: Arcos de petición en RAG

- Arcos de asignación. P_i tiene asignado un recurso de tipo R_1. (ver figura 1.4)

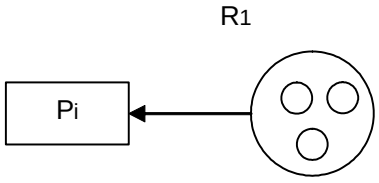

Figura 1.4: Arcos de asignación en RAG

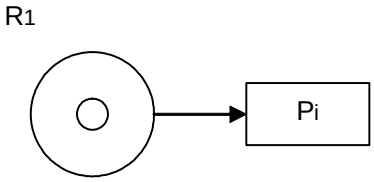

Figura 1.5: Arcos de producción en RAG

- Arcos de producción (ver figura 1.5)

  Para WFG el único nodo que existe es el que representa el proceso o transacción. Por ejemplo el gráfico figura 1.6, representa: T_2 tiene el candado del recurso compartido "x", T_1 esta esperando el dato "x" y T_1 no puede obtener el candado hasta que T_2 lo libere

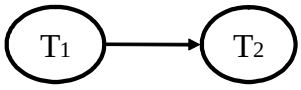

Figura 1.6: Ejemplo de WFG

1.1. Planteamiento del problema

Los candados mortales son un estado poco deseable en cualquier sistema. Por esto es necesario tener un método para evitarlo y en caso de llegarse a dar, poder detectarlo y restablecer todas las

transacciones que se encuentren bloqueadas.

Uno de los problemas que se pretende resolver en esta tesis, es analizar a priori un conjunto de transacciones que se desean ejecutar concurrentemente. La herramienta que utilizamos para modelar y analizar candados mortales son las redes de Petri. Se analizará la matriz de incidencia de la red de Petri para detectar si las transacciones pueden terminar su ejecución satisfactoriamente (entonces se ejecutarán las transacciones). En caso contrario, implementar una estrategia para que las transacciones puedan terminar su ejecución satisfactoriamente.

1.2. Estado del Arte

En la mayoría de los algoritmos para la detección de candados mortales de la literatura pretenden encontrar ciclos, existen diferentes métodos los cuales mencionaremos a continuación.

Existen varios esquemas de control de concurrencia que aseguran la seriabilidad, los cuales o bien retrasan una operación o bien abortan la transacción que ha realizado la operación. Esta consecuencia de las técnicas de control de concurrencia nos arrojaría un estado inconsistente en cualquier sistema.

En las bases de datos centralizadas uno de los esquemas más utilizados es el de cerradura (locking), que reserva todos los accesos de los recursos para que otros no puedan acceder a ellos. Con esta estrategia se podría generar un abrazo mortal.

Un sistema de base de datos distribuida está constituido por diferentes sitios, cada uno de los cuales es un sistema centralizado. Este tipo de base de datos tiene otro tipo de problemas a tratar por ejemplo la replicación de datos, la ejecución de transacciones en paralelo en diferentes sitios, entre otros. De igual manera, esto lleva a una situación de candado mortal.

Independientemente del tipo de sistema existen diferentes modelos para solicitar recursos. Por ejemplo, una transacción podría requerir los siguientes recursos "(tabla a y tabla b) o tabla c". Los modelos que explicaremos son AND, OR y AND-OR:

El modelo AND permite que una transacción bloquee todos los recursos que ésta necesita. Por ejemplo en la figura 1.7 el hilo H_{11} tiene dos peticiones de recursos (H_{21}y H_{31}). En este caso, para que el hilo se active se necesita satisfacer sus requerimientos para que se pueda activar.

Otro modelo es el OR el cual necesita cualquiera de los recursos solicitados. Por ejemplo en la figura 1.7 el hilo H_{11} no está en candado mortal puesto que H_{21} puede realizar su ejecución. Por

último el modelo AND-OR el cual es una combinación de los modelos antes mencionados. La figura 1.7 muestra una situación de candado mortal, correspondiente al ciclo H_{11}- H_{31}-H_{33}- H_{43}- H_{41}- H_{11}

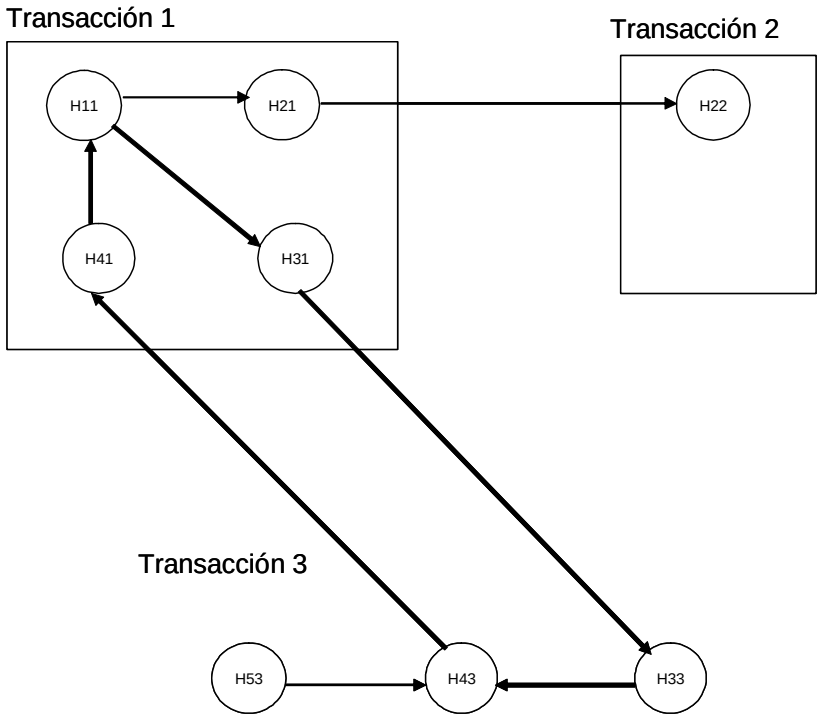

Figura 1.7: Ejemplo de grafica WFG.

Existen diferentes tipos de algoritmos para detectar los candados mortales. Uno de ellos es el trabajo realizado por Menasce y Muntz [6] el cual construye un gráfico como el de la figura 1.7 diseñado por cada proceso y enviado a todos los procesos vecinos. Cada proceso tiene la obligación de analizar el gráfico para detectar estructuras cíclicas.

Otro de los algoritmos interesantes es el de Mitchell y Meritt [16] el cual envía mensajes de pruebas. Este mensaje lo genera un proceso inicial y es enviado a los procesos vecinos; si el mensaje regresa, significa que existe un ciclo y por lo tanto un candado mortal.

Uno de los más utilizados es el trabajo desarrollado por Chandy y Lamport [15] el cual consiste en un historial de los eventos de todos los procesos. Es decir, todos los recursos que está utilizando

o que está requiriendo. Con el análisis del historial de las actividades de los procesos se puede detectar aquellas transacciones que se encuentran en candado mortal.

Observemos que la técnica WFG para la detección de abrazos mortales no es muy útil para las transacciones que se encuentran esperando por una dependencia de transacciones o de datos.

Los candados mortales han sido estudiados en la fase de dependencia de transacciones y de datos por Bertino, Chiola y Manzini [1]. Las dependencias de transacciones que definen son el commit y el abort. La dependencia de datos depende de las operaciones de lectura y escritura sobre datos compartidos. Además definen la siguiente sintaxis:

Tenemos un conjunto de transacciones τ y T una transacción en él. Además, TD un subconjunto que no incluye a T. Una dependencia entre T y una transacción de TD se define con la siguiente sintaxis:

$R1 < gendep > ::= < cdep > < adep >$

$R2{,}1 < cdep > ::= empty$

$R2{,}2 < cdep > ::= T \rightarrow < term >$

$R3{,}1 < adep > ::= empty$

$R3{,}2 < adep > ::= T \leftarrow < term >$

$R4{,}1 < term > ::= T\prime : T\prime \in \ TD$

$R4{,}2 < term > ::= < andterm >$

$R4{,}3 < term > ::= (< orterm >)$

$R5 < andterm > ::= T\prime : T\prime \in \ TDAND < term >$

$R6 < orterm > ::= T\prime : T\prime \in \ TDOR < term >$

Dónde:

<gendep> Especifica las dependencias generales, las cuales son <cdep> (dependencia commit) y <adep> (dependencia abort).

Existen además los términos AND y OR donde la única restricción es que el termino OR tiene que estar entre paréntesis.

Así una expresión correcta seria la que se muestra en la Figura 1.8:

Este lenguaje se encuentra definido en [1] se llama *multiform transaction model*. En este lenguaje, se especifica una secuencia de bloques coordinados, los cuales consisten en un grupo de transacciones con un orden parcial de ejecución, especificado en la cláusula de dependencias (using dependency).

```
void example_advanced()
{
 corrdinate;
cobegin
        begin_trans(T1) ... end_trans(T1)
        begin_trans(T2) ... end_trans(T2)
        ...
        begin_trans(Tn) ... end_trans(Tn)
coend;
 using dependency{
 T1 → (T2 OR T6)
  T2 → T3 AND T4
   }
}
```

Figura 1.8: Ejemplo de un conjunto de transacciones especificada con el lenguaje multiform transaction model.

El ejemplo especifica que T_1 no puede ejecutar commit hasta que T_2 o T_6 realice commit. A su vez T_2 no podrá ejecutar commit hasta que T_3 y T_4 terminen su ejecución.

Para poder detectar candados mortales con transacciones avanzadas Bertino, Chiola y Manzini proponen extender WFG para incluir la dependencia de transacciones. La cual la llaman *AND-OR graph*. El grafico correspondiente lo definen de la siguiente forma:

- Cada transacción T_i corresponde a un nodo en el *AND-OR graph*.
- Cada termino OR en la expresión de una dependencia general es representado por un nodo adicional en el *AND-OR graph*, se le llama pseudo nodo.
- La dependencia de abort $T_i \leftarrow T_j$ se representa como una dependencia commit $T_i \rightarrow T_j$ en el *AND-OR graph* puesto que los dos tienen la misma condición de espera. Por ejemplo T_i tendría que esperar que T_j aborte o ejecute commit en los casos anteriores.
- Cada termino AND en la expresión de una dependencia general es representado por un arco especial llamado arco-AND que une a las transacciones T'de acuerdo a la regla $R4{,}1y4{,}2$.

En la figura 1.9 se muestra el *AND-OR graph* de la figura 1.8. Para la expresión OR que

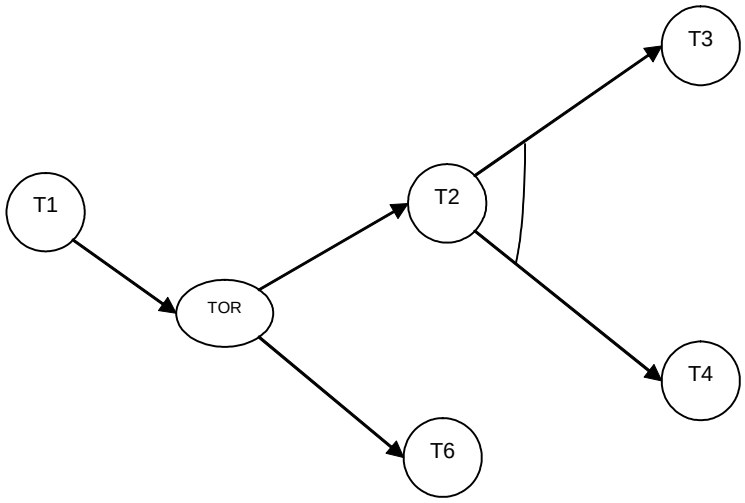

Figura 1.9: Extensión de WFG. $AND - ORGraph$

involucra a T_2 y T_6 se agrega un pseudo nodo. Para la expresión AND se agrega un arco especial (arc-AND) el cual une a T_3 y T_4.

Ya generado el *AND-OR graph* se tiene que analizar para detectar candados mortales. Para esto mapean el *AND-OR graph* a redes de Petri. El mapeo lo realizan de la siguiente forma:

1. Cada nodo (que representa una transacción) del *AND-OR graph* esta asociada con una transición en la PN .

2. Cada pseudo nodo esta asociada con un lugar especial.

3. Para cada arco-AND se define una transición adicional .

4. Si los nodos de la red no tienen asignado un pseudo nodo se define un lugar adicional y se conecta un arco con la transición que representa el nodo.

5. Para cada transición que representa un nodo, conectamos un arco de salida al lugar que representa la relación AND o OR.

A la PN generada le llaman *A-O net* y posee las siguientes características es ordinaria (el peso de sus arcos es 1), estructuralmente es persistente (cada lugar tiene al menos un arco de salida)

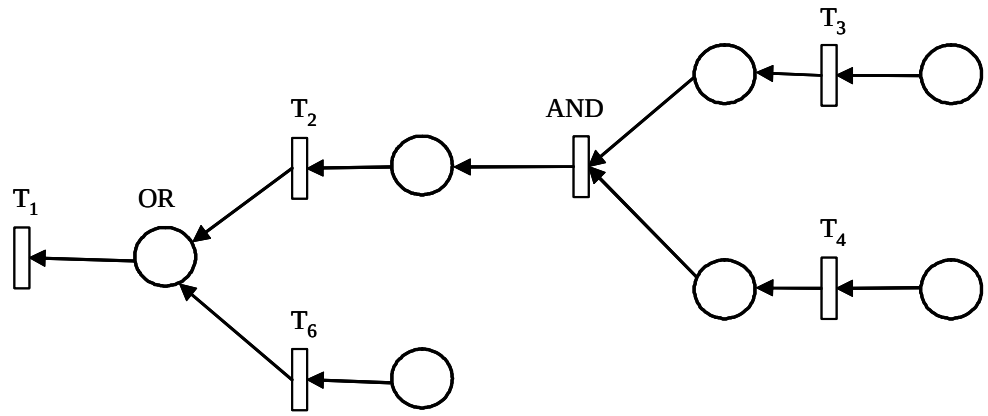

Figura 1.10: Ejemplo de una A-O net

y es libre-elección (un lugar tiene varios arcos de salida).La A-O net formada de la figura 1.9 se puede ver en la figura 1.10.

Las PN de libre-elección tienen la propiedad de caracterizar la ausencia o presencia de los candados mortales si cumple con:

Teorema 1.1 *Las AO nets están libres de candados mortales si y solo si no existen estructuras "sifón". Una transición que se encuentre en una estructura sifón nunca podrá habilitarse*

Definición 1.1 *Definición de sifón:*

$S \subseteq P : \forall t : t^{\circ} \cap S \neq 0,^{\circ} t \cap S \neq 0$

Esto quiere decir que existirá una estructura sifón si una transición no tiene lugar de entrada o de salida. Por lo tanto si una transición no tiene lugar de entrada no existirá forma de habilitarla, esto es la transición esta muerta.

El algoritmo funciona de la siguiente manera:

1. Inicia con el conjunto $S = P$
2. Si una transición existe sin arco de entrada, entonces S es una estructura sifón
3. Si una transición existe sin arco de entrada, entonces borrar la transición de la red junto con sus lugares de salida.

4. Cuando un lugar es borrado, también se deben borrar los arcos que se conectan

El algoritmo trabaja de forma recursiva. Así la PN resultante quedara libre de estructuras sifón, puesto que elimina transiciones. La PN resultante será libre de candados mortales.

En esta tesis desarrollamos un algoritmo más eficiente. La detección se realiza analizando la estructura de la PN para detectar aquellas transiciones que podrían tener conflicto.

1.3. Motivación

Los algoritmos convencionales para la detección de candados mortales no incluyen la dependencia de transacciones y de datos, por ende no pueden modelar y detectar candados mortales que surjan de esta situación.

En [1] se plantea la solución a este problema utilizando WFG. Sin embargo los autores comentan que necesitan extender WFG para poder abarcar las dependencias de transacciones. Ya que han modelado las transacciones en WFG mapean esta gráfica a redes de Petri donde realizan la detección de candados mortales.

En esta tesis se desarrollo un algoritmo que modela en redes de Petri la dependencia de transacciones. Las redes de Petri comparadas con WFG son una herramienta más eficiente y eficaz para modelar sistemas concurrentes. Puesto que se puede entre otras cosas modelar el dinamismo de las transacciones.

Otra de las desventajas de utilizar WFG como herramienta para modelar sistemas concurrentes es que no tiene como tal una herramienta matemática para analizar la estructura de la gráfica. Dado este problema Bertino, Chiola y Manzini [1] tienen que mapear WFG extendida a una red de Petri para poder detectar candados mortales. La detección que proponen se hace mediante estructuras sifón. En este trabajo se desarrollo un algoritmo para la detección analizando matemáticamente la estructura de la red de Petri. En otras palabras se generan todas las posibles rutas que pueden tomar las transacciones para terminar su ejecución satisfactoriamente. Siendo esta solución mas completa que la propuesta por [1]

1.4. Descripción del trabajo

En esta tesis proponemos modelar con PN las dependencias de un grupo de transacciones. El análisis lo hacemos en la estructura de la gráfica. Esto es, necesitamos en primera instancia detectar ciclos en la gráfica. En segundo lugar necesitamos detectar dos o más rutas que tengan el mismo inicio y el mismo final. El motivo de encontrar este tipo de rutas es detectar cuando varias transacciones acceden a un mismo dato.

La tesis presenta varias aportaciones:

1. Se propone un algoritmo para modelar con PN un conjunto de transacciones.
2. Se propone un método para modelar las operaciones de escritura y lectura en un dato compartido.
3. Se desarrolla un editor gráfico para la generación de una PN con un conjunto de transacciones dadas.
4. Un algoritmo recursivo para la búsqueda de rutas en una matriz de incidencia.
5. Se desarrolla un análisis de estas rutas para determinar la presencia o ausencia de candados mortales en un conjunto de transacciones.
6. Implementación de los análisis.

La escritura de la tesis se desarrollo de la siguiente manera:

En el capitulo 2 se presentan los fundamentos de las bases de datos, donde hablamos de las características principales y sus propiedad. Se trata el tema de control de concurrencia para garantizar la propiedad de seriabilidad. Además se mencionan dos técnicas utilizadas para el control de concurrencia los candados y la exclusión mutua, el problema de trabajar con estas técnicas es el posible resultado de los candados mortales. Qué es el tema al cual enfocamos esta tesis. Por ultimo mencionamos unas de las técnicas básicas para tratar los candados mortales.

En el capitulo 3 se describen los fundamentos teóricos de las PN, comenzando por su definición formal, las reglas de disparo de transiciones y el poder que tienen para modelar sistemas. Además, se presentan las propiedades que poseen y los métodos de análisis. Poniendo mayor énfasis en la

matriz de incidencia. Explicamos de igual manera de las extensiones que tienen las PN para poder modelar una mayor cantidad de sistemas.

En el capitulo 4 se presenta la sintaxis propuesta para las dependencias de transacciones y de datos, así como el diagrama de flujo, pseudo código de la aplicación realizada. Además se muestran algunos ejemplos con un conjunto de transacciones propuestas.

En este capitulo se detalla el comportamiento de la aplicación así como un estudio de los candados mortales modelados con redes de Petri.

En el capitulo 5 mencionamos el lenguaje de programación que se utilizo para implementar la plataforma de desarrollo. Se comenta el diseño, la arquitectura y las clases utilizadas.

Por ultimo en el capitulo 6 se explican los resultados obtenidos de este trabajo de investigación así como las limitaciones que presenta. Se enlistan los posibles trabajos futuros.

Capítulo 2

Introducción a Base de Datos y Candados Mortales

Los candados mortales se pueden presentar en diferentes sistemas (por ejemplo los sistemas operativos o de manufactura), en este trabajo estudiaremos aquellos que se presentan en bases de datos. Para esto es necesario dar una introducción a los manejadores de base de datos y dar énfasis en las propiedades de las transacciones.

Un tema muy importante y que da origen al problema que estamos tratando es el control de concurrencia en esta sección mencionamos lo mas comunes y aquellos que podrían originar un candado mortal.

Por último mencionamos el tema que nos compete los candados mortales enfocados en cuatro vertientes la detección, el análisis, la recuperación y como los manejadores de base de datos han trabajado este tema.

2.1. Manejadores de base de datos.

Un sistema manejador de base de datos (DBMS, DataBase Management System), consiste en un conjunto de datos relacionados entre sí y un grupo de programas para tener acceso a esos datos. Al conjunto de datos se conoce como base de datos. El objetivo primordial de un DBMS es crear un ambiente en que sea posible guardar y recuperar información de la base de datos en forma

conveniente y eficiente.

Los sistemas de base de datos se diseñan para manejar grandes cantidades de información. El manejo de los datos incluye tanto la definición de las estructuras para el almacenamiento de la información como los mecanismos para el manejo de la información. Además, el sistema de base de datos debe cuidar la seguridad de la información almacenada en la base de datos, tanto como las caídas del sistema como contra los intensos de acceso no autorizado. Si los datos van a ser compartidos por varios usuarios, el sistema debe evitar la posibilidad de obtener resultados anómalos[2].

A los usuarios de la base de datos se les debe permitir varias operaciones dentro de las cuales se encuentran[3]:

- Adicionar nuevos archivos a la base de datos.
- Insertar nuevos datos en los archivos existentes.
- Recuperar datos de los archivos existentes.
- Actualizar datos de los archivos existentes.
- Borrar datos de los archivos existentes.
- Remover archivos de la base de datos.

El sistema de base de datos debe permitir los puntos antes mencionados, pero al cumplirlos se pueden encontrar con varios problemas, donde los más sobresalientes los mencionamos a continuación:

- **Redundancia e inconsistencia de datos**. Cuando un dato esta repetido varias veces entramos en redundancia. Si se actualiza uno de estos datos, se deben actualizar todos los repetidos, si no se hace de esta manera tenemos un sistema inconsistente.
- **Dificultad para el acceso de datos**. El ambiente debe permitir recuperar la información requerida en forma conveniente y eficiente.
- **Aislamiento de los datos**. Varias transacciones se pueden ejecutar concurrentemente; el sistema debe garantizar que todas se ejecuten satisfactoriamente.

- **Usuarios múltiples**. Permitir a varios usuarios trabajar concurrentemente. En un sistema de este tipo puede existir inconsistencia en la información.
- **Problemas de seguridad**. No es recomendable que todos los usuarios del sistema de base de datos puedan tener acceso a toda la información.
- **Problemas de integridad**. Los valores de los datos que se almacenan en la base de datos deben satisfacer ciertos criterios de consistencia.

2.2. Modelo entidad-relación

Para describir la estructura de una base de datos es necesario definir el concepto de modelo de datos. El modelo de datos es un grupo de herramientas para describir los datos, sus relaciones, su semántica y sus limitantes. Existen varios grupos pero en esta tesis solo nos concentraremos en el modelo entidad-relación (E-R).

Se escogió este modelo por ser uno de los más aceptados en el diseño de base de datos. El modelo E-R se basa en la percepción de un mundo real que consiste en un conjunto de objetos básicos llamados entidades, y de las relaciones entre estos objetos. La distinción se logra asociando a cada entidad un conjunto de atributos que describen al objeto.

La estructura lógica de una base de datos puede expresarse gráficamente por medio de un diagrama E-R que consta de los siguientes componentes:

- Rectángulos, que representan entidades.
- Elipses, que representan atributos.
- Rombos, que representan relaciones entre las entidades.
- Líneas, que conectan a los atributos con las entidades y viceversa.

Por ejemplo, consideremos un sistema bancario que consta de clientes y sus cuentas. El diagrama E-R correspondiente se muestra en la figura 2.1

Los datos y sus relaciones se representan por medio de una serie de tablas, cada una de las cuales tiene varias columnas con nombres únicos.

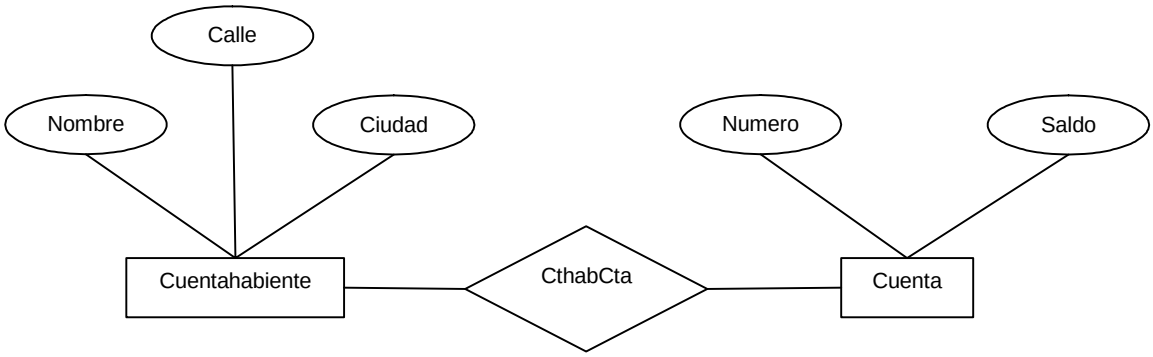

Figura 2.1: Ejemplo de un diagrama E-R.

2.3. Estructura general de un sistema de base de datos.

Un sistema de base de datos se divide en módulos que se encargan de cada una de las tareas del sistema general. Así, el diseño de la base de datos debe incluir una consideración de la interface entre el sistema de base de datos y el sistema operativo. La estructura esta organizada como sigue:

- **Manejador de archivos**, es el encargado de asignar espacio en el disco y de las estructuras de datos que se van a emplear para representar la información almacenada.
- **Manejador de base de datos**, es la interface entre los datos de bajo nivel almacenados en la base de datos, los programas de aplicación y las consultas que se hacen al sistema.
- **Procesador de consultas**, traduce las proposiciones en lenguaje de consulta a instrucciones de bajo nivel que puede entender el manejador de base de datos.
- **Precompilador DML (Data Manipulation Language)**, interactúa con el procesador de consultas para generar el código apropiado.
- **Compilador DDL (Data Definition Language**), convierte las proposiciones DDL en un conjunto de tablas que contienen metadatos (datos acerca de los datos). Tales tablas se almacenan en el diccionario.

La figura 2.2 muestra los componentes y las conexiones entre ellos.

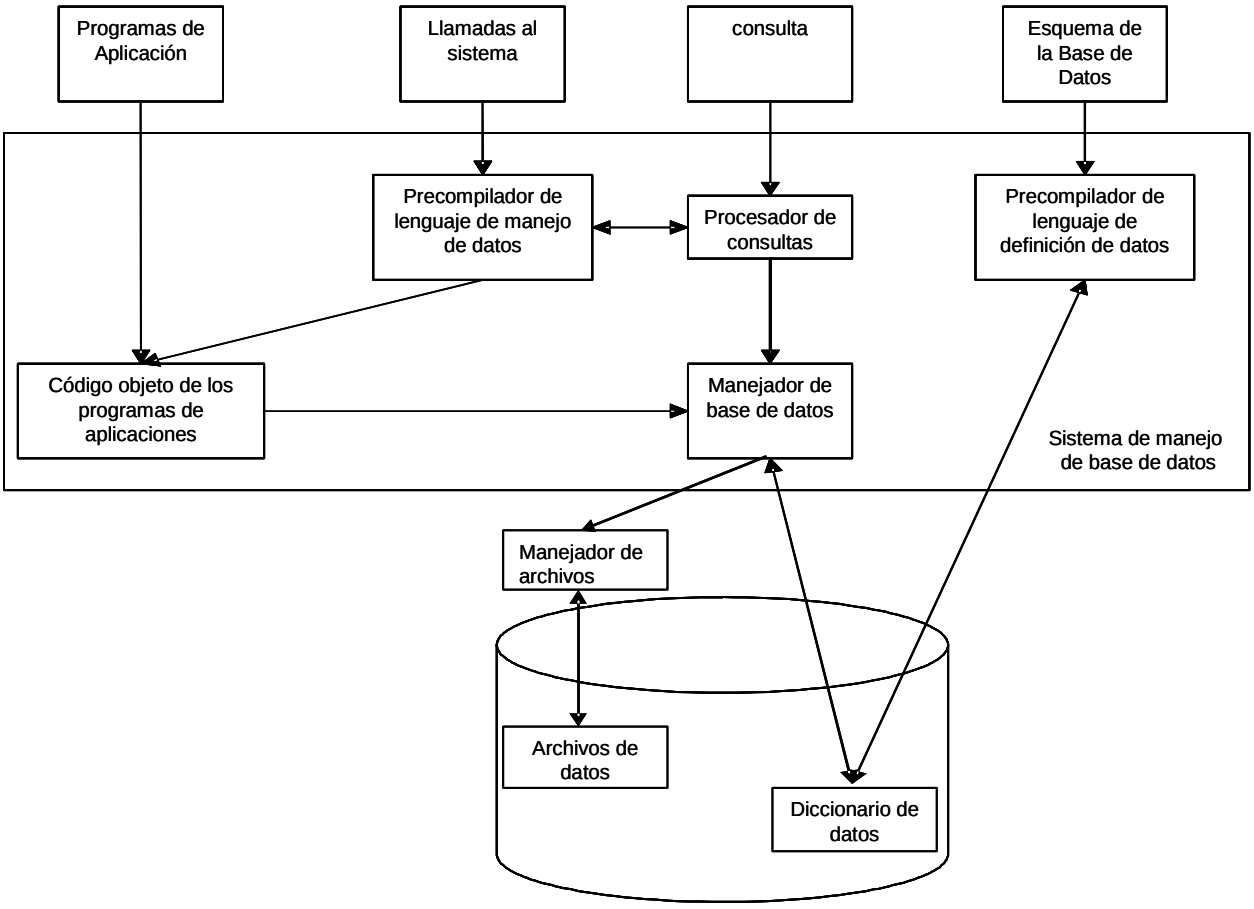

Figura 2.2: Estructura de un sistema de base de datos

2.4. Lenguaje SQL

Los lenguajes mas sobresalientes para consultar las bases de datos son SQL, Quel y QBE. Solo vamos a mencionar SQL por ser uno de los más utilizados en base de datos comerciales.

La consulta en SQL (Structured Query Language) se compone de tres cláusulas: *select* (elige), *from* (de) y *where* (donde).

- La cláusula select sirve para listar todos los atributos que se desean en el resultado de consulta.
- La cláusula from es una lista de relaciones que se van a examinar durante la ejecución de la expresión.
- La cláusula where se compone de un predicado que incluye atributos de las relaciones que aparecen en la cláusula from.

La consulta tiene la siguiente forma:

`select` A_1`,` A_2`, ...,` A_n

`from` r_1`,` r_2`,...` r_n

`where P`

Las A's representan atributos, las r's representan las relaciones y P es un predicado. El SQL forma el producto cartesiano de las relaciones que se nombran en la cláusula *from*; realiza una selección utilizando el predicado de la cláusula *where* y proyecta el resultado a los atributos de la cláusula *select*.

La eliminación de tuplas es muy sencilla. Una solicitud de eliminación se expresa en forma muy parecida a una consulta, con la diferencia que las tuplas elegidas ya no se muestran al usuario sino que se borran de la base de datos. Sólo se pueden quitar tuplas completas. En SQL, una eliminación se expresa así:

```
delete from r
where p
```

P representa un predicado y r representa una relación. Las tuplas t en r para las cuales P(t) se cumpla serán eliminados.

Para insertar datos en una relación debe especificarse la tupla que se va a insertar, o bien, debe escribirse una consulta cuyo resultado sea el conjunto de tuplas que se insertará. Obviamente, los

valores de los atributos de las tuplas que se inserten deben pertenecer al dominio de esos atributos. Además, las tulpas que se inserten deben ser del mismo orden.

```
insert into r
values (d1, d2, ... dn)
insert into r
select A1, A2, ..., An
from r1, r2,... rm
    where P
```

Existen situaciones en las que se desea cambiar el valor en una tupla sin cambiar todos los valores de ésta. Si se realizan estos cambios empleando la eliminación o inserción, es posible que no se puedan conservar los valores que no se desean modificar. En vez de ello se usa la proposición *update* (actualizar). Como en el caso de la inserción y la eliminación, pueden escogerse las tuplas que se van a actualizar empleando una consulta.

```
update r
set An= dn
```

o

```
update r
set An= dn
where P
```

2.5. Transacciones

Una colección de varias operaciones en una base de datos es llamada transacción. Estas operaciones pueden ser de consulta (por ejemplo, *SELECT*) o para modificar los registros (por ejemplo, *UPDATE*)[2].

En base de datos la transacción es una primitiva básica y debe de ser consistente y confiable. En la Figura 2.3 consideramos una transacción sencilla para entender su comportamiento[3]. En primer lugar tenemos *START TRANSACTION* que indica el inicio de la transacción. A continuación colocamos las operaciones que tendrá que realizar la transacción (por ejemplo actualizar una tabla). Si la transacción ejecuta exitosamente todas sus operaciones. Entonces ejecuta *COMMIT*. Esto significa que las modificaciones se hacen permanentes.

Si llega a existir algún error al ejecutarse la transacción. Entonces se ejecuta *ROLLBACK*. Esto es que todas las modificaciones hechas por la transacción se deshacen.

```
START TRANSACTION
        UPDATE... SELECT... .
IF error ROLLBACK
ELSE COMMIT
```

Figura 2.3: Una transacción sencilla

El modelo de una transacción se muestra en la Figura 2.4, al inicio de la transacción la base de datos esta en un estado confiable, durante la ejecución el estado de la base de datos se comporta de manera inestable y al finalizar se debe garantizar que el estado de la base de datos sea confiable.

Concepto formal de una transacción:

Definición 2.1 *Sea $O_{ij}(x)$ una operación O_j de la transacción T_i la cual trabaja sobre alguna entidad x. $O_j \in \{read, write\}$ y O_j es una operación atómica. Es decir se ejecuta como una unidad invisible. Se denota por $OS_i = \cup\ j\ O_{ij}$ al conjunto de todas las operaciones de la transacción T_i. También, de denota por N_i la condición de terminación para T_i, donde, $N_i \in \{abort, commit\}$.*

2.6. Control de concurrencia

Cuando varios usuarios intentan modificar datos al mismo tiempo, es necesario establecer controles para impedir que las modificaciones de un usuario influyan negativamente en las de otros. El sistema mediante el cual se controla lo que sucede en el sistema se denomina control de concurrencia, en esta sección mencionaremos los más utilizados en la literatura.

2.6.1. Teoría de la seriabilidad

Una calendarización (schedule) se define sobre un conjunto de transacciones $T = \{T_1, T_2, \ldots, T_n\}$ y especifica un orden de la ejecución de las operaciones de las transacciones. La calendarización se puede especificar como un orden parcial sobre el conjunto T.

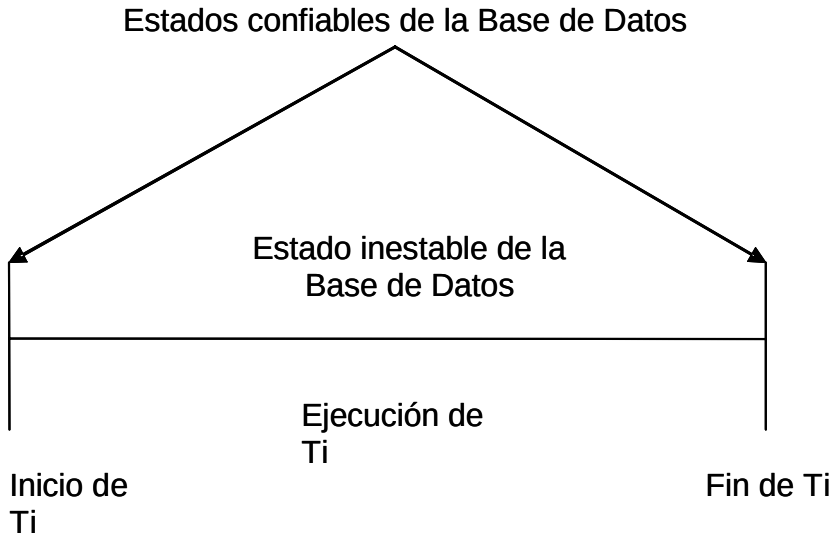

Figura 2.4: Modelo de transacción

Por ejemplo, consideremos las siguientes transacciones que se encuentran en [2]:

```
T0read(A);
A:= A-50;
write (A);
read (B);
B:= B+50;
write (B);
```

```
T1 read (A);
temp:= A*0.1
A:= A-temp
write (A);
read (B);
B:= B+temp;
write (B);
```

Imaginemos que A = 1000 y B = 2000. Si estas transacciones se ejecutaran en forma aislada y secuencialmente (primero T_0 y después T_1). El valor final de A y B será de 855 y 2145 respec-

tivamente. De esta forma la base de datos se encontraría en un estado estable y consistente. Las secuencias de ejecución pueden variar puesto que se pueden ejecutar varias transacciones concurrentemente y en un mismo procesador.

Si la ejecución fuera la siguiente:

```
T0          T1
read (A)
A:= A-50
            read (A)
            temp:=A*0.1
            A:=A-temp
            write(A)
            read(B)
write(A)
read(B)
B:=B+50
write(B)
            B:=B+temp
            write(B)
```

Los valores finales de A y B fueran 950 y 2100. Por lo tanto la base de datos se encontraría en un estado inconsistente. Una forma de definir un sistema concurrente correctamente, es requerir que el resultado de procesar un conjunto de transacciones sea el mismo que el que se produciría si se ejecutaran esas transacciones en serie. Se dice que un sistema que cumpla esta propiedad garantiza la seriabilidad.

Los algoritmos de control de concurrencia deben sincronizar la ejecución de transacciones concurrentes para garantizar el aislamiento de ellas.

El propósito del control de concurrencia es intervenir en el acceso concurrente a un objeto. Con esto se pretende no ver comprometida la consistencia de los datos.

Con el control de concurrencia avalamos la seriabilidad en bases de datos. Existen tres enfoques[17]:

- **Espera**. Si dos transacciones entran en conflicto, una de ellas tiene que esperar hasta que la otra termine su ejecución.

	Lectura (x)	Escritura(x)
Lectura (x)	Verdadero	Falso
Escritura(x)	Falso	Falso

Figura 2.5: Compatibilidad de candados

- **Timestamp (hora de entrada)**. Cuando hay conflicto de transacciones, el sistema le asigna un tiempo de ejecución a cada transacción.
- **Rollback**. Si dos transacciones entran en conflicto, las acciones realizadas por las transacciones se deshacen y se reinicia la transacción.

2.6.2. Algoritmo basado con el mecanismo de espera

Para implementar este tipo de algoritmos el sistema tiene que proveer de candados en las entidades de la base de datos. Cuando una transacción accede un dato tiene que obtener el candado de este. Si está disponible, se le asigna y sólo ésta transacción puede "entrar" a modificar el recurso, en caso de no estar disponible la transacción tendrá que esperar a que se libere el candado.

Existen dos modos para poner un candado:

1. **Candado de lectura**. La transacción obtiene el candado de manera compartida. Cualquier otra transacción que desee leer el mismo dato puede obtener el candado.
2. **Candado de escritura**. La transacción obtiene el candado de manera exclusiva. Ninguna transacción puede obtener el candado de lectura o escritura de ese dato.

Después de que la transacción termina su operación tiene que liberar los candados. Supongamos que T_i solicita un candado de escritura en el dato x, en el cual T_j tiene un candado de lectura. ¿Puede concederse a la transacción T_i el candado de x? para responder esto, tenemos que ver la compatibilidad de los candados la cual se muestra en la figura 2.5

Se debe exigir que todas las transacciones del sistema sigan una serie de reglas, llamadas protocolos para candados, que indiquen cuando una transacción puede poner o quitar un candado.

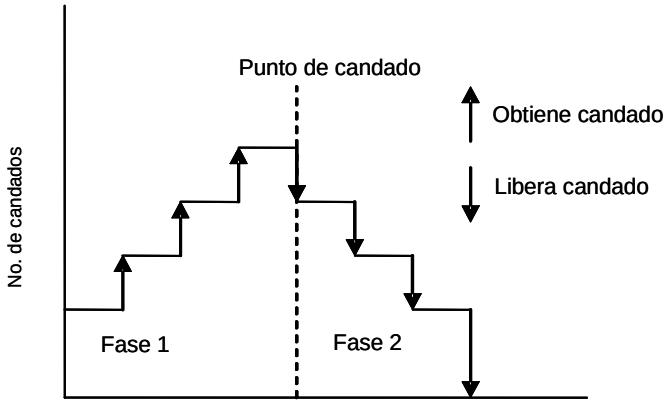

Figura 2.6: Protocolo de dos fases para candados

2.6.3. Protocolo de dos fases (2PL) para candados.

Este protocolo requiere que todas las transacciones hagan sus solicitudes para poner y quitar candados en dos fases: en la primera fase se obtienen todos los candados, se le llama fase de crecimiento. Cuando se liberan los candados se dice que entro en la fase de encogimiento y no puede obtener mas candados. Las dos fases se ilustran en la figura 2.6..

Otras extensiones que se basan en este algoritmo son:

Candados centralizados. Se designa un sitio primario de la red donde se almacenan los candados de las tablas de toda la base de datos. Con la finalidad de otorgar los candados a las transacciones.

Candados descentralizados. El administrador de los candados es compartido por todos los sitios de la red. La ejecución de las transacciones envuelve la participación y la coordinación de más de un sitio.

2.6.4. Algoritmo basado con el mecanismo de timestamp.

En este método se determina el orden de seriabilidad a priori. A cada transacción se le asigna un tiempo de entrada único. Obviamente el tiempo debe estar sincronizado en todos los nodos del sistema distribuido. Lamport describe un algoritmo para sincronizar todos los relojes distribuidos, pasándose mensajes. Si un mensaje llega a un nodo local de un nodo remoto con un tiempo mayor.

El nodo local incrementa su tiempo hasta igualarse con el nodo remoto. De esta forma todos los relojes avanzan hasta sincronizarse.

El algoritmo trabaja de la siguiente manera. A cada transacción T_j del sistema se le asocia una hora de entrada fija, denotada por $TS(T_j)$, es asignada antes de iniciar su ejecución. Las horas de entrada de las transacciones determina el orden de serialibilidad. El orden esta rígida por la siguiente regla:

Teniendo dos operaciones en conflicto O_{ij} y O_{kl} pertenecientes, respectivamente a las transacciones T_i y T_k. O_{kl} se ejecutara primero si y solo si $TS(T_i) < TS(T_k)$. En este caso se dice que T_i es una transacción mas vieja que T_k [10].

Otro algoritmo es el de Thomas. En el cual, los nodos del sistema distribuido se comunican entre si para votar por cada transacción. La transacción que más votos tenga es a la que se le brindaran todos los recursos para terminar su ejecución.

2.6.5. Algoritmo basado con el mecanismo de rollback.

La idea de este algoritmo es validar los resultados de las transacciones antes de ejecutar commit. Si la validación falla, la transacción se repite y trata validarse nuevamente. Las transacciones tienen tres fases de vida, dependiendo si la transacción es de consulta o de actualización:

1. **Fase de lectura**. En esta fase se ejecuta la transacción. Se leen los valores de los datos y se almacenan en variables locales. Todas las operaciones de escritura se realizan sobre variables temporales, sin actualizar la base de datos.

2. **Fase de validación**. Se realiza una prueba de validación para determinar, si se pueden copiar con éxito las variables temporales a la base de datos sin causar violación a la seriabilidad.

3. **Fase de escritura**. Si la transacción logra la validación, se aplican las actualizaciones a la base de datos. En caso contrario retrocede.

2.7. Candados mortales.

Cualquier algoritmo para el control de concurrencia basado en candados puede dar origen a los candados mortales. Un ejemplo de este problema se encuentra en [1]:

Supongamos que las transacciones T_i y T_j son tareas dentro de la misma transacción y tenemos una dependencia especificada en que, el commit de T_i debe preceder al de T_j. Imaginemos que T_j adquiere el candado de escritura en el dato "x" y después T_i requiere adquirir el candado de lectura de "x". Obviamente T_j no puede liberar el candado hasta que realiza el commit; sin embargo, T_j debe esperar al commit de T_i. T_i, por otro lado, está esperando que T_j libere el candado en "x". Por lo tanto T_i y T_j esperaran indefinidamente.

2.7.1. Modelos de candados mortales.

Hay cuatro condiciones para que un candado mortal ocurra:

- **Exclusión mutua**. Si un recurso ha sido adquirido por 1 o mas transacciones, se le niega el acceso a cualquier otro.
- **No apropiación**. Los recursos son liberados voluntariamente, ningún proceso puede forzar a liberar los recursos (inclusive los del sistema Operativo)
- **Almacena y espera**. Una transición que tiene asignado un recurso, puede solicitar más y esperar a que le sean concedidos.
- **Espera circular.** Puede existir un conjunto de transacciones tal que T_0 este esperando por un recurso que tiene T_1, y éste esté esperando un recurso que tiene $T_2, \ldots$ T_{n-1} esta esperando un recurso que tiene T_n y T_n esta esperando un recurso que tiene T_0.

Existen cuatro tipos de solicitudes de recursos:

Modelo de solo un recurso (single resource)

En este modelo la transacción solo puede obtener un candado a la vez. Es decir, si una transacción requiere el candado de un dato tiene que esperar a que se le otorgue para poder solicitar otro recurso.

Modelo AND.

En el modelo AND, se permite a las transacciones requerir un conjunto de recursos. La transacción se bloquea hasta que se le entregan todos los recursos solicitados. Por ejemplo una requisición de (a and b and c). Requiere asignarle a la transacción los tres recursos.

Modelo OR.

El número de requisiciones a un grupo de recursos se satisface otorgando cualquier recurso solicitado. Por ejemplo una requisición (a or b or c). La transacción requiere que se le asigne cualquiera de estos tres recursos.

Modelo AND-OR

Este modelo es una combinación de los mencionados anteriormente. Por ejemplo la requisición (a and (b or c)). La transacción requerirá de a y b o c para finalizar su ejecución.

En general los candados mortales tienen tres facetas la prevención, la detección y la recuperación.

2.7.2. Prevención de candados mortales

Este método garantiza que los candados mortales no ocurran. Uno de los métodos más sencillos, es requerir que cada transacción obtenga todos los candados de los recursos compartidos que necesite. Si los recursos están disponibles la transacción finalizara su ejecución. En caso contrario tendrá que esperar a que se liberen. La mayor desventaja de este método es que una transacción tarde un tiempo indefinido en finalizar su ejecución.

Otro método es imponer un orden parcial a los datos y permitir que las transacciones obtengan sus candados con este orden utilizando un protocolo de árbol. Un enfoque diferente es utilizar una hora de entrada (timestamp) para darle prioridad a las transacciones y resolver los candados mortales abortando aquellas transacciones con mayor tiempo. Existen dos esquemas propuestos para este enfoque:

- **Esperar y morir (WAIT-DIE)**. Cuando la transacción T_i solicita un candado que T_j tiene en ese momento, a T_i se le permitirá esperar si y solo si T_i es mas antigua que T_j. En caso contrario T_i abortará y reiniciará con la misma hora de entrada.
- **Herir y morir (WOUND-WAIT)**. Cuando la transacción T_i solicita un candado que T_j tiene en ese momento, a T_i se le permitirá esperar si y solo si T_i es mas joven que T_j. En caso contrario T_j abortará y el candado lo obtendrá T_i.

Estos esquemas evitan que una transacción espere por un tiempo indefinido. La diferencia entre estos dos métodos es que *WAIT-DIE* prefiere a las transacciones jóvenes, puesto que a las antiguas las mata. Por otro lado *WOUND-WAIT* prefiere a las transacciones antiguas. Donde las transacciones tendrán que esperar hacerse viejas para su ejecución.

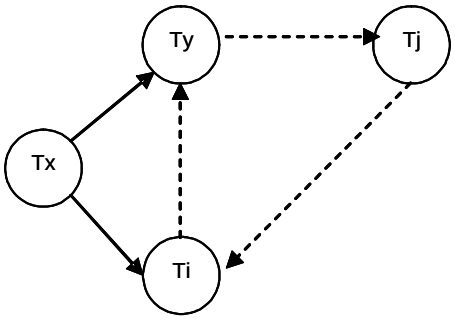

Figura 2.7: Ejemplo de WFG

2.7.3. Detección y recuperación de candados mortales

Si un sistema no garantiza la ausencia de candados mortales, debe emplearse un esquema de detección y recuperación. Para lograrlo el sistema debe:

- Mantener información de las asignaciones de los datos a las transacciones, así como solicitudes pendientes.
- Contar con un algoritmo que utilice esta información para detectar candados mortales
- Recuperarse si el algoritmo detecta un candado mortal.

Una herramienta utilizada para analizar candados mortales es wait for graph (WFG). WFG es un gráfico dirigido que representa las relaciones de espera entre las transacciones. Los nodos del gráfico representan las transacciones concurrentes del sistema. El arco $T_i \rightarrow T_j$ en WFG significa que T_i espera a que T_j libere un candado en algún dato compartido. Utilizando WFG se tienen que encontrar ciclos en la gráfica para determinar si existe un candado mortal.

En [2] se muestra el siguiente ejemplo para entender WFG. Imaginemos que la transacción T_i solicita un dato que T_j tiene. Se agrega un arco entre los nodos (T_i, T_j) a el gráfico, lo que resulta un nuevo estado del sistema. Representado en la gráfica 2.7

Esta gráfica contiene un ciclo formado por T_y, T_j y T_i(los arcos punteados).

Cuando se detectan ciclos en la gráfica, el sistema debe recuperarse de este problema. La forma más sencilla de romper el ciclo es escoger una o más transacciones del grupo que se encuentra en

candado mortal y abortarlas.

Para determinar que transacción tendrá que abortarse es necesario elegir aquella que tenga un "costo mínimo", este termino tiene diferentes factores por tomar en cuenta:

1. El tiempo que lleva ejecutándose la transacción así como el tiempo que le falta por terminar.
2. Cuantos datos ha utilizado la transacción.
3. Cuantos datos mas necesita para que la transacción termine
4. Cuantas transacciones están involucradas.

Algoritmos para la detección de candados mortales en sistemas de base de datos distribuidos.

Los algoritmos para detección de candados mortales se dividen en: path-pushing, basado en pruebas y detección de estado global.

Path-Pushing En este algoritmo Menasce y Muntz mantienen WFG como base para la detección. Cada sitio de la red colecciona periódicamente las transacciones que están en estado de espera para construir el gráfico, busca ciclos en él, y si los detecta los resuelve (rompe el ciclo abortando algunas transacciones). El gráfico resultante lo envía a los sitios vecinos. El nodo que recibe el gráfico, fusiona estos y detecta ciclos en él; y así consecutivamente. El algoritmo termina cuando un sitio tiene un gráfico de la situación global de la red y esta en condiciones de anunciar un candado mortal.

Basado en pruebas Cuando una transacción T_i requiere un candado que lo tiene otra transacción, T_i tiene que iniciar una prueba. Por ejemplo T_i le manda un mensaje de prueba a la transacción que lo tiene esperando, a su vez esta transacción manda un mensaje a las transacciones que lo tienen en espera y así sucesivamente. Si el mensaje retorna a T_i se declara un candado mortal.

Detección de estado global. Todo el trabajo relacionado a este tipo de algoritmos lo inicio Chandy y Lamport. Donde la idea principal es contar con un WFG fiable. Para esto se cuenta con un administrador de transacciones (TM) que tiene el control de todas las transacciones del sistema, además maneja todos los accesos a los datos compartidos. Con esto cada vez que una transacción

desee acceder a algún dato o que espere por un candado se lo tendrá que reportar al TM. Con toda la información controlada el gráfico generado nos dirá cuando realmente existe un candado mortal.

2.8. Manejo de candados en DBMS

En los últimos años, el software de bases de datos ha experimentado un auge extraordinario. No es extraño pues, que existan varios gestores de bases de datos, programas que permitan manejar la información de manera más sencilla.

En general los Sistemas Manejadores de Base de Datos (DBMS) deben facilitar las siguientes funciones:

- Almacenar físicamente los datos.
- Garantizar la consistencia y la integridad de los datos.
- La ejecución de las transacciones se realicen de manera atómica.
- Manejar vistas de la información.

En esta tesis mencionaremos como manejan su control de concurrencia y sobre todo como manejan los candados mortales. Dentro de los DBMS mas famosos encontramos: Oracle^{TM}, Borland Interbase^{TM} , MySQL^{TM} y PostgreSQL^{TM}, entre otros. De los cuales daremos una breve reseña a continuación.

2.8.1. MySQL

En la versión MySQL 4.0[22] se pueden escoger entre tres tipos de tablas (MyISAM, ISAM, HEAP) y recientemente se agregaron InnoDB y BDB (Base de datos Berkeley). Cuando se crea una tabla se tiene que indicar el tipo de tabla (el default es MyISAM).

MySQL tiene entre su sintaxis LOCK TABLES y ULOCK TABLES. La finalidad de colocar candados es emular transacciones y obtener más velocidad cuando se actualizan las tablas. Se garantiza que esta técnica es libre de candados mortales. Sin embargo resulta poco confiable puesto que si la operación se ejecuta y existe un error no hay forma de deshacer los cambios hechos por esta.

BDB e InnoDB son ambas tipos de tablas transaccionales. Esto es, se puede utilizar la sentencia estándar BEGIN seguida de varias consultas y finalizar con un COMMIT o ROLLBACK para completar la transacción. O, se pueden correr en modo AUTOCOMMIT, así que cada consulta es una transacción separada.

Las ventajas de usar las tablas que soporten transacciones es que se pueden combinar varias operaciones, es mas seguro y puedes ejecutar ROLLBACK si fallan algunas actualizaciones

Sin embargo no garantizan que las transacciones estén libres de candados mortales. Mencionan algunos trucos para tratar con este problema.

- Utilizar SHOW INNODB STATUS para determinar la causa del último candado mortal
- Estar preparados para rehacer una transacción que falla por causa de un candado mortal
- Realizar commit a las transacciones periódicamente
- Acceder las tablas o columnas con un orden establecido
- Utilizar lo menos posible los candados para las tablas
- Serializar las transacciones
- Y por ultimo implementar semáforos para el control de concurrencia

2.8.2. PostgreSQL.

A diferencia de la mayoría de otros sistemas de bases de datos que usan bloqueos para el control de concurrencia, PostgreSQL[25] mantiene la consistencia de los datos utilizando un modelo multiversión. Esto es, que mientras se consulta una base de datos, cada transacción ve una imagen de los datos (una versión de la base de datos) como si fuera en el pasado, sin tener en cuenta el estado actual de los datos. Esto evita que la transacción vea datos inconsistentes que pueden ser causados por la actualización de otra transacción concurrente en la misma fila de datos, proporcionando aislamiento transaccional para cada sesión de la base de datos.

PostgreSQL ofrece varios modos de bloqueo para controlar el acceso concurrente a los datos en tablas. Algunos de estos modos de bloqueo los adquiere PostgreSQL automáticamente antes de la ejecución de una declaración, mientras que otros son proporcionados para ser usados por

las aplicaciones. Todos los modos de bloqueo (excepto para AccessShareLock) adquiridos en un transacción se mantienen hasta la duración de la transacción.

AccessShareLock

Un modo de bloqueo adquirido automáticamente sobre tablas que están siendo consultadas. PostgresSQL libera estos bloqueos después de que se haya ejecutado una declaración.

Conflictos con AccessExclusiveLock:

RowShareLock

Adquirido por **SELECT FOR UPDATE** y **LOCK TABLE** para declaraciones **IN ROW SHARE MODE**.

Entra en conflictos con los modos ExclusiveLock y AccessExclusiveLock.

RowExclusiveLock

Lo adquieren **UPDATE, DELETE, INSERT y LOCK TABLE** para declaraciones I**N ROW EXCLUSIVE MODE.**

Choca con los modos ShareLock, ShareRowExclusiveLock, ExclusiveLock y AccessExclusive-Lock.

ShareLock

Lo adquieren **CREATE INDEX y LOCK TABLE** para declaraciones **IN SHARE MODE.** Está en conflicto con los modos RowExclusiveLock, ShareRowExclusiveLock, ExclusiveLock y Ac-cessExclusiveLock.

ShareRowExclusiveLock

Lo toma **LOCK TABLE** para declaraciones **IN SHARE ROW EXCLUSIVE MODE**. Está en conflicto con los modos RowExclusiveLock, ShareLock, ShareRowExclusiveLock, Exclu-siveLock y AccessExclusiveLock.

ExclusiveLock

Lo toma **LOCK TABLE** para declaraciones **IN EXCLUSIVE MODE.**

Entra en conflicto con los modos RowShareLock, RowExclusiveLock, ShareLock, ShareRowEx-clusiveLock, ExclusiveLock y AccessExclusiveLock.

AccessExclusiveLock

Lo toman **ALTER TABLE, DROP TABLE, VACUUM y LOCK TABLE.**

Choca con RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, Exclusive-

Lock y AccessExclusiveLock.

Nota: Sólo AccessExclusiveLock bloquea la declaración SELECT (sin FOR UPDATE).

2.8.3. Oracle.

Cuando un candado mortal ocurre en algún nodo o en algún cluster, el administrador de candados de Oracle[23] lo puede detectar con un mensaje de error escrito (Oracle error message: ORA-00060). El archivo de alerta generado no contiene suficiente información sin embargo menciona que el error es generado por la aplicación no por Oracle. Es importante mencionar que Oracle no detecta problemas generados en un sistema distribuido.

La manera en que Oracle detecta los candados mortales es cuando dos transacciones están esperando por un mismo recurso por demasiado tiempo. En ese instante manda el error 00060 y genera un archivo de alerta. El administrador de la base de datos para romper el candado tiene que matar una de las transacciones o todas.

Oracle propone una serie de scripts (archivos con alguna secuencia de comandos) que al realizar su ejecución genera información de las transacciones que están en estado de espera y las peticiones a los recursos. Con esta información el administrador de la base de datos podrá tomar una mejor decisión para romper el candado mortal.

2.8.4. Borland Interbase™

Interbase[24] garantiza la consistencia de su base de datos utilizando el mecanismo multiversión. El cual se basa en generar diferentes versiones de los registros, asociándolos cada uno con la transacción que los modifica. El rol de las transacciones en este mecanismo es vital para garantizar la concurrencia y la fiabilidad de la base de datos.

Cada vez que se inserta un registro en una base de datos InterBase se almacena, junto con los datos, una referencia a la transacción a la que está asociada la operación de escritura. A su vez, cada transacción (sea de escritura o lectura) se le asocia un identificador (número entero). Este identificador sigue una secuencia ascendente, de tal forma que las transacciones más antiguas tienen identificadores más bajos.

Cuando se modifica un registro, se genera una nueva versión del registro asociada a la transacción que lo ha modificado. Esta nueva versión mantiene un enlace a la versión antigua del registro y,

recíprocamente, la versión antigua mantiene un enlace a la nueva, formando una especie de cadena.

Cuando se genera la nueva versión del registro, se compara con la versión antigua para generar un bloque con las diferencias encontradas, denominado BDR. Este registro con las diferencias entre versiones se almacena en una nueva localización en la base de datos, y la versión nueva del registro se sitúa en el lugar que ocupada la antigua versión.

De esta forma se garantiza que solo se modificarán los campos que realmente han cambiado. De tal forma que no se almacena por cada versión una copia completa del registro, sólo de los campos que hayan sido cambiados por la operación de actualización.

En las eliminaciones, el BDR almacena la versión antigua del registro, mientras que la nueva versión simplemente almacena la referencia a la transacción que eliminó el registro y una marca de borrado.

Para controlar la concurrencia de las transacciones Interbase sigue las siguientes reglas:

- Si el identificador de la transacción es menor que el identificador de la transacción de la última versión del registro, entonces la transacción no puede ver o modificar el registro
- Si el identificador de la transacción es igual que el identificador de la transacción de la última versión del registro, entonces la transacción sí puede ver y/o modificar el registro
- Si el identificador de la transacción es mayor que el identificador de la transacción de la última versión del registro, y además esta última fue confirmada (commit) antes de que la transacción comience, entonces ésta sí puede ver y/o modificar el registro

La detección de candados mortales en Interbase se hace mediante tablas temporales. Al ejecutarse las transacciones se llenan estas tablas. Con el comando SHOW SYSTEM nos despliega toda la información de la tabla, hay que tener en mente el número de la transacción.

En la tabla temporal existe un campo llamado TMP$TRANSACTION_DEADLOCKS en cual es un entero que nos muestra el numero de transacciones que están en candado mortal.

2.9. Comentarios finales

En este capitulo se presentaron argumentos teóricos para la comprensión de cómo interactúan las transacciones concurrentes en una base de datos. Mencionando los mecanismos de control de

concurrencia que garantizan propiedades importantes para las base de datos (entre algunos el aislamiento). Sin embargo, presenta ciertas desventajas el uso de ellas, donde la que se trato en profundidad es el candado mortal.

Además se menciono como se ha tratado este problema y que técnicas han utilizado para poder evitarlo, detectarlo y solucionarlo.

Esta base teórica nos permite comprender ampliamente el problema y poder generar una solución más eficiente. El siguiente capitulo lo dedicamos a las Redes de Petri que será la herramienta utilizada para trabajar con los candados mortales.

Capítulo 3

Fundamentos de PN

Las Redes de Petri (PN) son una herramienta gráfica y matemática que se puede aplicar a cualquier sistema o área y pueden, entre otras cosas, describir gráficamente la concurrencia de las actividades del sistema; por lo tanto podemos simular y modelar la concurrencia en recursos compartidos.

Las PN fueron inventadas por el alemán Karl Adam Petri en 1962. En su tesis doctoral "kommunikation mit automaten"(Comunicación con autómatas) en la facultad de Matemáticas y Física de la Universidad Darmstadt en Alemania. En este trabajo, K.A. Petri estableció las bases para una teoría de comunicación entre componentes asíncronos de un sistema de cómputo. Definió una herramienta matemática de propósito general para describir las relaciones que existen entre condiciones y eventos[4].

Las PN presentan dos características interesantes. Primero hacen posible modelar y visualizar el comportamiento de la concurrencia, la sincronización y los datos compartidos. En segundo los resultados teóricos son plenamente confiables.

3.1. Definiciones Preliminares

La PN esta compuesta de 4 partes: un conjunto de lugares P, un conjunto de transacciones T, una función de entrada I y una función de salida O. Las funciones de entrada y salida establecen una conexión entre los lugares y las transiciones. La función de entrada I mapea de una transición

t_j a una colección de lugares $I(t_j)$, conocidos como lugares de entrada de la transición. La función de salida O mapea a una transición t_j a una colección de lugares $O(t_j)$ conocidos como lugares de salida de la transición.

La estructura de una PN, es una 4-tupla

$$C = (P, T, I, O)$$

$P = \{p_1, p_2, \ldots, p_n\}$ es un conjunto finito de lugares $n \geq 0$.

$T = \{t_1, t_2, \ldots, t_m\}$ es un conjunto finito de transiciones $m \geq 0$.

El conjunto de lugares y transiciones debe cumplir que $P \cap T = \emptyset$.

Un lugar p_i es un lugar de entrada a la transición t_j si $p_i \in I(t_j)$; p_i es un lugar de salida si $p_i \in O(t_j)$.

Ejemplo 3.1 *Veamos la siguiente estructura de una PN:*

$C = (P, T, I, O)$
$P = \{p_1, p_2, p_3, p_4, p_5\}$
$T = \{t_1, t_2, t_3, t_4\}$
$I(t_1) = \{p_1\}$
$I(t_2) = \{p_2, p_3, p_5\}$
$I(t_3) = \{p_3\}$
$I(t_4) = \{p_4\}$
$O(t_1) = \{p_2, p_3, p_5\}$
$O(t_2) = \{p_5\}$
$O(t_3) = \{p_4\}$
$O(t_4) = \{p_2, p_3\}$

De acuerdo a esta estructura la transición t_1 tiene como lugares de entrada p_1 ($I(t_1) = \{p_1\}$) y tendrá como lugares de salida a p_2 ,p_3 y p_5 ($O(t_1) = \{p_2, p_3, p_5\}$). De acuerdo a esto se construye la gráfica que se muestra en la Figura 3.1.

Informalmente la PN esta constituida de dos tipos de nodos, los lugares y las transiciones. Un lugar se representa por un círculo y una transición por un rectángulo. Los lugares y las transiciones se conectan por un arco. El número de lugares es finito y no debe ser cero. De igual manera pasa con las transiciones. El arco conecta lugares con transiciones y transiciones con arcos.

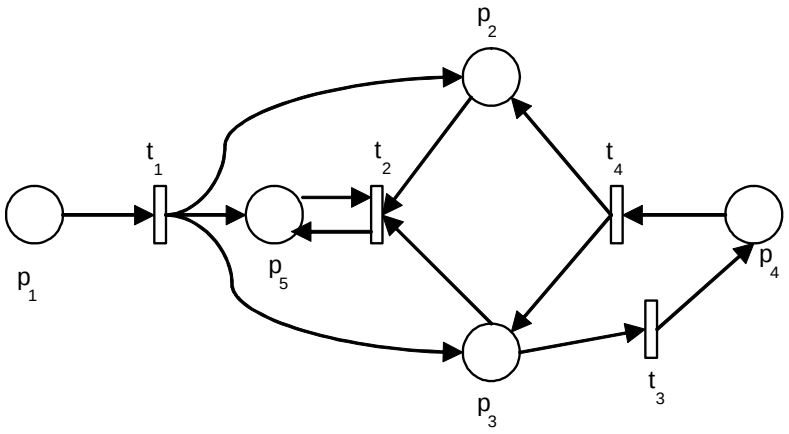

Figura 3.1: Red de Petri del ejemplo 3.1

Una marca es la asignación de un token a un lugar de la PN. El token se representa por un punto en el circulo, que como ya mencionamos representa un lugar. El número de tokens contenidos en un lugar p_i se denota como $\mu(p_i)$ o μ_i. Por ejemplo en la Figura 3.2 tenemos lo siguiente $\mu_1 = 1$, $\mu_2 = 1$, $\mu_3 = \mu_4 = 0$.

Formalmente una marca se define como:

Definición 3.1 *Una marca μ en una PN $C = (P, T, I, O)$ es una función de un conjunto de lugares P a números enteros no negativos N. $\mu : P \rightarrow N$.*

Una PN marcada $CM = (C, \mu)$ es una estructura de una PN $C = (P, T, I, O)$ y una marca μ. También se puede definir como $CM = (P, T, I, O, \mu)$

La Figura 3.2 muestra una PN con una marca inicial $\mu_0 = (1, 1, 0, 0)$.

La ejecución de una PN esta controlada por el número y la distribución de tokens. Los tokens residen en los lugares y controlan la ejecución de las transacciones en la red. La PN se ejecuta disparando las transiciones. Cuando se dispara la transición se remueven los tokens del lugar de entrada y se crean nuevos tokens en el lugar de salida.

Una transición puede dispararse si esta habilitada, para habilitarse necesita que los lugares de entrada tengan al menos un token. Por ejemplo en la Figura 3.3 se muestra el disparo de la transición t_1. Como se puede ver antes del disparo de la transición: el lugar p_1 tiene dos tokens y p_2

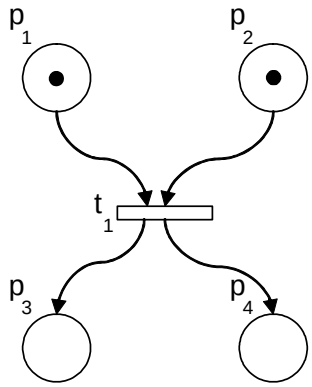

Figura 3.2: Una PN marcada

tiene uno. Al dispararse la transición se remueve un token tanto de p_1 como de p_2, y se les agrega un token a los lugares p_3 y p_4.

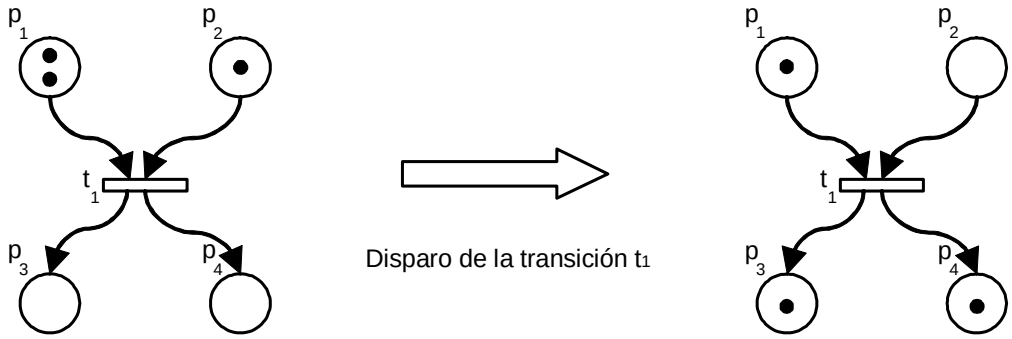

Figura 3.3: Disparo de transicione t_1

Las PN tienen dos conceptos primitivos: eventos y condiciones. Los eventos son acciones que toman lugares en el sistema. La ocurrencia de estos eventos es controlado por el estado del sistema. El estado del sistema puede describir un conjunto de condiciones. Una condición es una descripción lógica del estado del sistema. La condición por lo tanto puede ser verdadera o falsa.

Puesto que los eventos son acciones, estos pueden ocurrir. Para que un evento ocurra, se hacen necesarias ciertas condiciones. Estas son las llamadas precondiciones del evento. La ocurrencia del

evento puede causar otras condiciones, que son las llamas postcondiciones.

Ejemplo 3.2 *Modelemos una máquina que vende refrescos. La máquina esta en espera hasta que un cliente llega y ordena un refresco, después la máquina atiende la orden y entrega el pedido.*

Las condiciones del sistema son:

- La máquina esta esperando
- Una orden llega y espera
- La máquina trabaja en la orden
- Se termina la orden

Los eventos pueden ser:

1. Llega una orden
2. La máquina empieza a trabajar en la orden
3. La máquina termina de trabajar en la orden
4. se entrega la orden

Las precondiciones y las postcondiciones se muestran en la siguiente tabla.

Evento	Precondición	Postcondición
1	Ninguna	b
2	a,b	c
3	c	d,a
4	d	Ninguna

Dónde:

a La máquina esta esperando un cliente

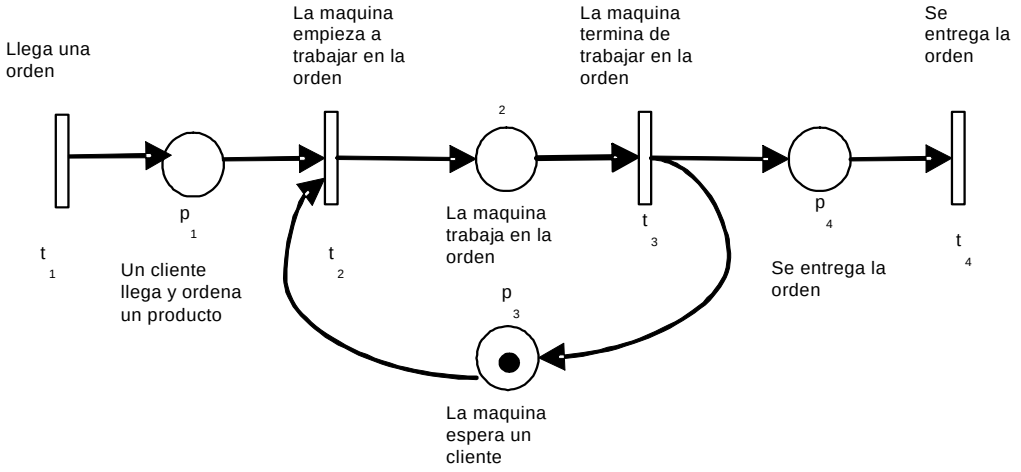

Figura 3.4: Modelación de una máquina de refrescos

b Un cliente llega y ordena un producto

c La máquina trabaja en la orden

d Se entrega la orden

La modelación ahora resulta muy sencilla. Las condiciones son modeladas como lugares y los eventos son modelados como transiciones. Las entradas de las transiciones son las precondiciones correspondientes al evento; las salidas son las postcondiciones. La ocurrencia de un evento corresponde al disparo de la transición correspondiente. El dinamismo de la PN la dan los tokens. Esto es cuando un cliente llega y ordena su refresco, se dispara la transición t_1 y coloca un token en el lugar p_1. De esta forma se inicia la simulación y el estado donde se encuentre el sistema será donde se encuentre el token. En la figura 3.4 se muestra la PN resultante.

3.2. Propiedades de PN

Dos propiedades que se estudian con los modelos de PN son aquellas que dependen de la marca inicial y aquellas que son independientes o que analizan la estructura de la PN. A continuación mencionaremos las propiedades más importantes en la bibliografía.

Seguridad (safeness).

Un lugar en una PN es seguro si el número de tokens en el lugar nunca excede uno. Una red de Petri es segura si todos los lugares de la red son seguros.

Formalmente se define:

Definición 3.2 *Una PN* $C = (P, T, I, O, \mu_0)$ *es segura si cada lugar en* P *es seguro.*

Las PN son seguras por definición, puesto que ninguna transición puede dispararse a menos que todos los lugares de salida están vacíos. Un lugar p_i que se obliga a ser seguro debe ser suplementado por otro lugar p_i'. Las transacciones que utilizan p_i como entrada o salida deben ser modificadas como sigue:

If $p_i \in I(t_j)$ y $p_i \notin O(t_j)$, entonces adiciona p_i' a $O(t_j)$

If $p_i \in O(t_j)$ y $p_i \notin I(t_j)$, entonces adiciona p_i' a $I(t_j)$

El objeto de este nuevo lugar p_i' es representar la condición p_i esta vació. De tal forma que p_i y p_i' son complementarios; si p_i tiene token p_i' no debe tener token y viceversa.

Para muchos sistemas pasar de un lugar seguro a otro seguro es vital, por ejemplo los sistemas de hardware. Sin embargo no todos los sistemas requieren de esta propiedad.

Limites (boundness)

La seguridad es un caso especial para la propiedad de los límites. Un lugar es k-seguro o k-limitado si el número de tokens en este lugar no excede el entero k.

Se dice que un lugar p es k-limitado si el número de tokens en p siempre es menor o igual a k (k es un número entero no negativo) para cada marca μ_i alcanzables desde la marca inicial μ_0.

Conservación

La cantidad de recursos que aparecen en un sistema real generalmente es un número fijo, por lo tanto, el número de tokens en un modelo de PN para este sistema debe permanecer igual sin importar las transiciones que se disparen. Cuando las PNs son usadas para representar sistemas de asignación de recursos la conservación es una propiedad importante.

Definición 3.3 *Una PN marcada* $C = (P, T, I, O, \mu_0)$ *es conservativa si para todo* $\mu \in R(C, \mu_0)$

$$\sum_{p_i \in P} \mu(p_i) = \sum_{p_i \in P} \mu_0(p_i)$$

Ejemplo 3.3 *Se puede mostrar que si el numero de entradas a una transición es igual al numero de salidas,* $\mid I(t_j) \mid = \mid O(t_j) \mid$. *Es decir, el disparo de la transición* t_j *no cambia el número de tokens. En la figura 3.5 se muestra una PN que no tiene la propiedad de conservación. Esto es porque si se disparan las transiciones* t_1 *y* t_2 *el número de tokens se reduce en* 1. *Si se disparan las transiciones* t_3 *y* t_4 *se aumenta un token.*

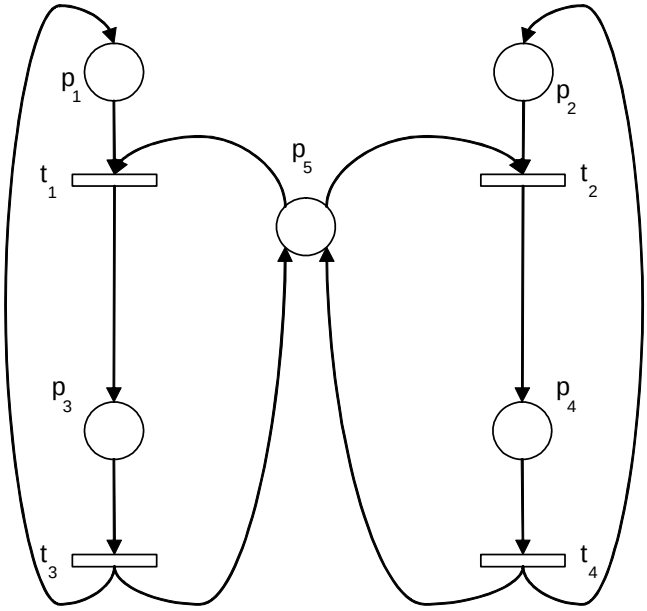

Figura 3.5: PN que no es conservativa

La Figura 3.6 muestra una PN que si cumple con la propiedad de conservación. Veamos que los tokens se mantienen constantes.

3.2.1. Vida.

El concepto de vida esta muy relacionado a la ausencia de candados mortales. Una PN (C, μ_0) se dice que tiene vida si para alguna marca alcanzable μ_i, es posible disparar al menos una transición en la red a través de alguna secuencia de disparo. Esto significa una PN viva garantiza la ausencia

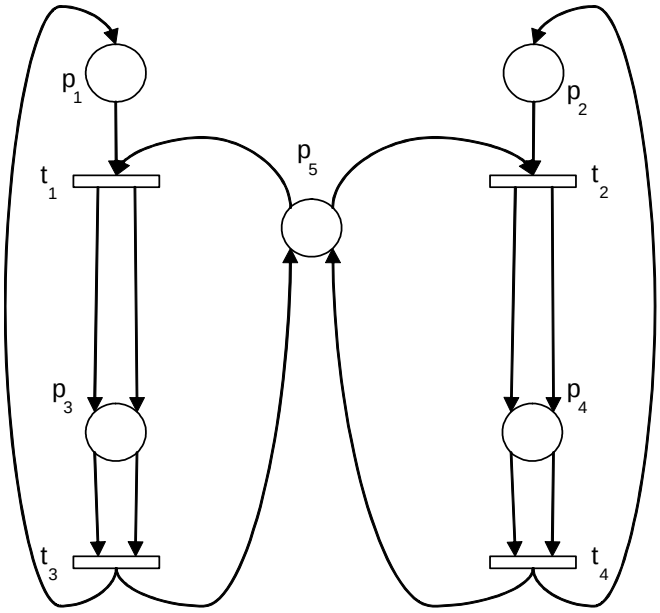

Figura 3.6: PN estrictamente conservativa

de candados mortales sin importar la secuencia de disparo. Una candado mortal en PN es una transición que nunca se dispara. Una transición esta viva si es potencialmente disparable por una marca μ_i.

Existen varios conceptos relacionados con la vida de una PN y están considerados para el estudio de candados mortales. Estos están definidos de la siguiente manera:

- Nivel 0: una transición t_j esta viva en el nivel 0 si nunca se dispara
- Nivel 1: una transición t_j esta viva en el nivel 1 si es potencialmente disparable. Es decir, si existe una marca $\mu_i \in R(C, \mu_i)$ tal que t_j es habilitada en μ'.
- Nivel 2: una transición t_j esta viva en el nivel 2 si para un entero n existe una secuencia de disparo en la cual t_j se dispara al menos n veces.
- Nivel 3: una transición t_j esta viva en el nivel 3 si t_j se dispara infinitamente en alguna secuencia de disparo.

- Nivel 4: si t_j es L1-viva (potencialmente disparable) en cada marca μ de $R(\mu_0)$.

Una transición que tenga nivel 0 esta muerta. Una transición que esta en el nivel 4 esta viva.

Ejemplo 3.4 *Como un ejemplo de estos niveles de vida consideremos la Figura 3.7. La transición* t_0 *nunca se va a disparar esta muerta. La transición* t_1 *se dispara exactamente una vez; por lo tanto es de nivel* 1*. La transición* t_2 *se puede disparar un número arbitrario de veces, pero este número de veces es dependiente al número de veces que se dispara* t_3*. Por lo tanto* t_2 *es de nivel* 2*. La transición* t_3 *puede dispararse un número infinito de veces por lo tanto es de nivel* 3*.*

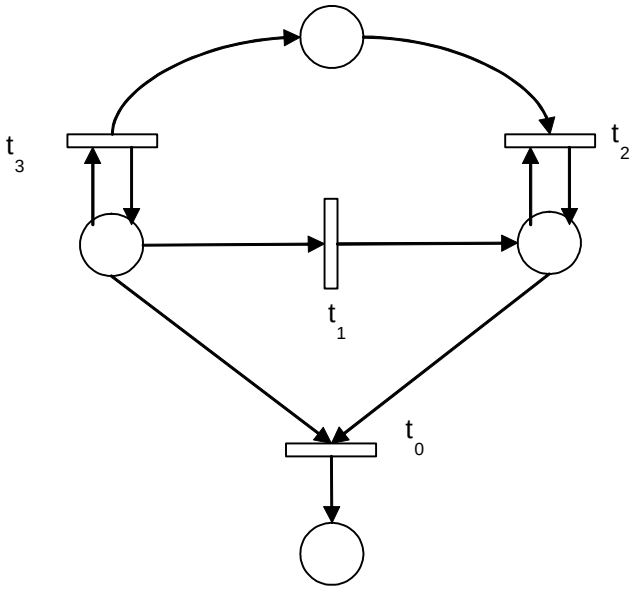

Figura 3.7: PN que ilustra los niveles de vida

3.2.2. Alcanzabilidad

La alcanzabilidad es la base para estudiar las propiedades dinámicas de un sistema. Para detectar si el sistema modelado alcanza un estado específico, es necesario encontrar una secuencia de disparo de transiciones que transforme una marca μ_i en una marca μ_j, donde μ_j representa el estado al que se quiere llegar y la secuencia de disparos representa el comportamiento funcional solicitado.

Ejemplo 3.5 *Consideremos la PN de la Figura 3.8. La marca inicial de esta PN es* $\mu_0 = (2, 0, 0, 1)$ *y se desea encontrar una secuencia de disparo para alcanzar la marca* $\mu_i = (1, 1, 1, 1)$*. Si se dispara la transición* t_1*, la cual tiene las precondiciones para dispararse, se podrá alcanzar* μ_i*. Veamos la Figura 3.9. Donde ya se disparo la transición. Como podemos ver la marca* μ_i *es alcanzable desde* μ_0*.*

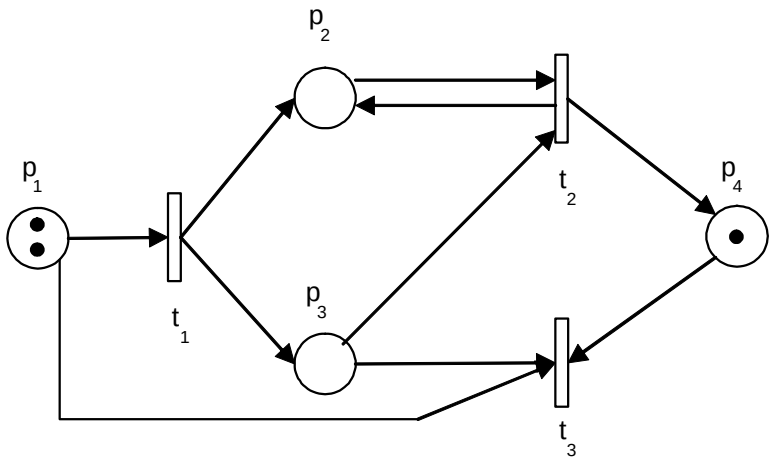

Figura 3.8: PN con marca inicial (2,0 ,0 ,1)

3.3. Análisis de PN

Las técnicas que se han utilizados en la literatura para analizar PN son el árbol de alcanzabilidad (*reachability tree*) y la matriz de incidencia. El primer método esencialmente enumera todas las marcas que se puedan alcanzar en una alcanzables PN. El segundo método trabaja con una matriz que contiene la estructura de la PN y en base a ella se puede estudiar al dinamismo de la PN.

3.3.1. Árbol de alcanzabilidad.

Dada una PN (C, μ_0), de la marca inicia μ_0 se pueden obtener tantas nuevas marcas como el numero de transiciones habilitadas. Para cada nueva marca podemos obtener nuevas marcas. Este

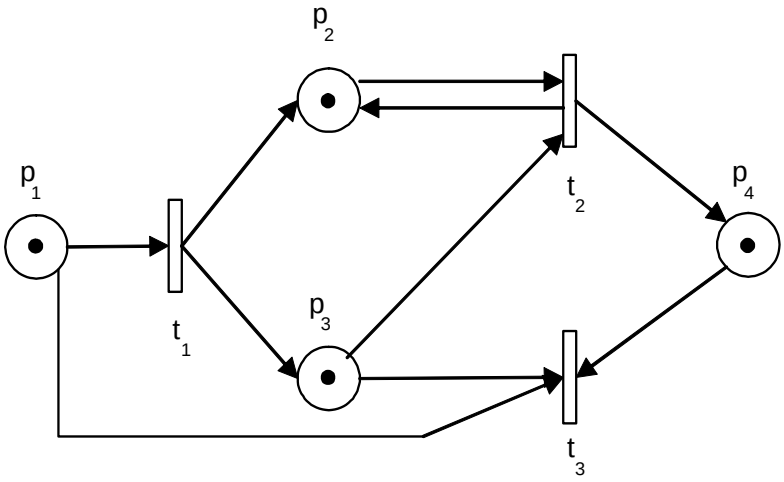

Figura 3.9: PN con marca (1, 1, 1, 1)

proceso se representa por medio de un árbol de marcas. Los nodos representan marcas generadas de μ_0 (la raíz) y sus sucesores, y cada arco representa el disparo de una transición, con la cual se transforman las marcas.

El árbol de alcanzabilidad se construye bajo el siguiente algoritmo:

1. Etiquetar la marca inicial μ_0 como la raíz del árbol y marcarla como "nueva".

2. Mientras existan marcas "nuevas", hacer lo siguiente:

   2.1 Seleccionar una marca nueva μ.

   2.2 Si μ es idéntica a una marca en la ruta desde la raíz hasta μ, entonces marcar a μ como "vieja" e ir a otra marca nueva.

   2.3 Si la marca μ no tiene transiciones activadas, rotular a μ como "nodo muerto".

   2.4 Mientras existan transiciones activadas en μ, para cada transición activada t en μ hacer lo siguiente:

   2.4.1 Obtener la marca μ' que resulta del disparo de t en μ;

2.4.2 En la ruta desde la raíz hasta μ si existe una marca μ" tal que $\mu'(p) \geq \mu"(p)$ para cada lugar p y $\mu' \neq \mu"$, es decir, μ" está contenido en μ', entonces reemplazar $\mu'(p)$ por ω para cada p tal que $\mu'(p) > \mu"(p)$.

2.4.3 Incluir μ' como un nodo, dibujar un arco con la etiqueta t desde μ a μ' y marcar a μ' como "nueva".

Ejemplo 3.6 *Tomemos como ejemplo la PN de la figura 3.10. La marca inicial es* $(1, 0, 0)$. *Con esta marca inicial dos transacciones se pueden habilitar* t_1 *y* t_2. *Se definen dos nuevos nodos del árbol de alcanzabilidad, los cuales se forman por el disparo de ambas transiciones. Los arcos se etiquetan con el nombre de las transiciones respectivas, iniciando en la marca inicial y terminando en las nuevas marcas. Ver Figura 3.11.*

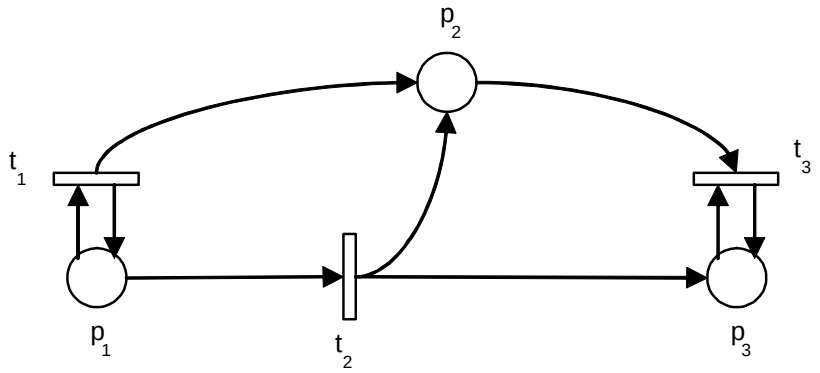

Figura 3.10: un PN para ilustrar el árbol de alcanzabilidad

Ahora consideramos las marcas alcanzables. De la marca $(1, 1, 0)$, podemos disparar t_1 y alcanzamos la marca $(1, 2, 0)$ y de t_2 alcanzamos $(0, 2, 1)$; de $(0, 1, 1)$ podemos disparar t_3 obteniendo $(0, 0, 1)$. Esto produce la Figura 3.12. Con las tres nuevas marcas repetimos el proceso, produciendo nuevas marcas que tendremos que adicionar al árbol. Con la marca $(0, 0, 1)$ no habilita ninguna transición por lo tanto esta muerta. Las dos marcas restantes se pueden ver en la figura 3.13.

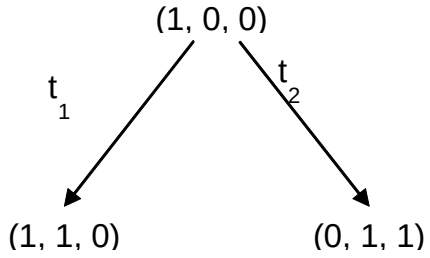

Figura 3.11: Primer paso para construir el árbol de alcanzabilidad

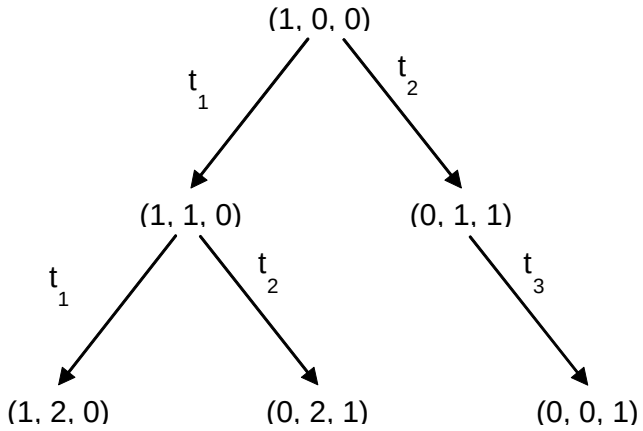

Figura 3.12: Segundo paso para construir el árbol de alcanzabilidad

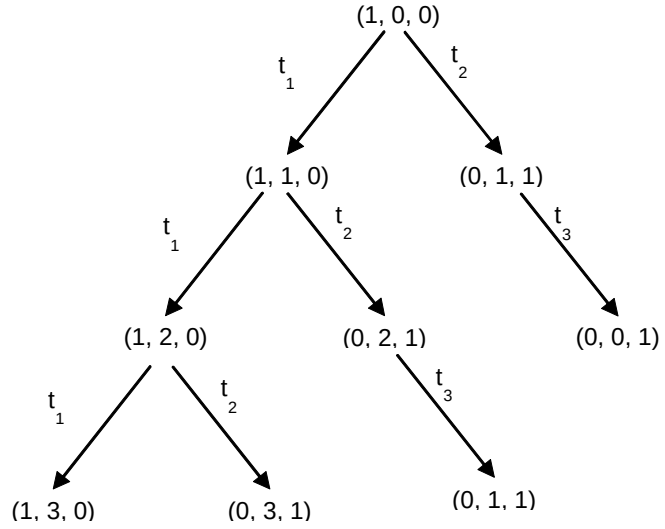

Figura 3.13: Tercer paso para construir el árbol de alcanzabilidad

3.3.2. Matriz de incidencia

La matriz de incidencia contiene información sobre la estructura de la red de Petri (independientemente de las marcas).

Definición 3.4 *Para una PN, con n transiciones y m lugares, la matriz de incidencia* $A = [a_{ij}]$ *es una matriz de números enteros de* $n \times m$*. El valor para cada elemento de la matriz está dado por:*$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$

donde $a_{ij}^{+} = w(i,j)$ es el peso del arco que conecta una transición $t_i \in T$ con su lugar de salida $p_j \in P$ y $a_{ij}^{-} = w(j,i)$ a es el peso del arco que conecta una transición $t_i \in T$ con su lugar de entrada $p_j \in P$.

En resumen a_{ij} representa la relación entre las transiciones y los lugares. Básicamente la forma de llenar la matriz A esta dada por:

$$a_{ij} = \begin{cases} -1 & \text{El lugar } p_j \in P \text{ es un lugar de entrada a la transición } t_i \in T. \\ 0 & \text{No existe un arco que conecta al lugar } p_j \in P \text{ con la transición } t_i \in T \text{ y vice versa.} \\ 1 & \text{El lugar } p_j \in P \text{ es un lugar de salida de la transición } t_i \in T. \end{cases}$$

Ejemplo 3.7 *La matriz de incidencia que se muestra en la figura 3.15 corresponde a la PN de la figura 3.14. Las filas se asocian con los lugares y las columnas se asocian con las transiciones. Es decir, la primera columna significa que* t_1 *tiene como entradas a* p_1 *y* p_2*. Por lo tanto las coordenadas* $(0,0)$ *y* $(0,1)$ *tienen valor de* -1*. Por otro lado la transición tiene como salidas a* p_3 *y* p_4*. Teniendo como valor 1 en las coordenadas* $(0,2)$ *y* $(0,3)$*.*

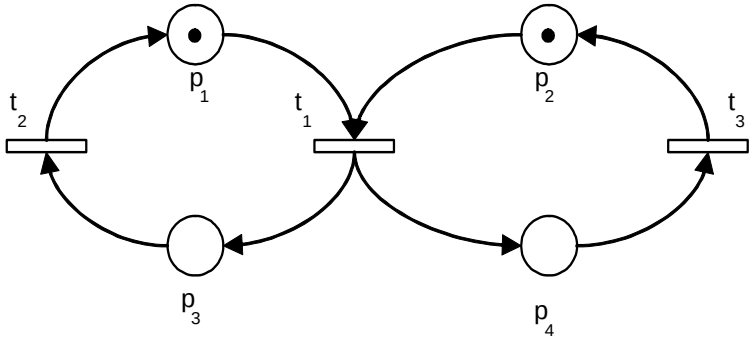

Figura 3.14: un PN para ilustrar la matriz de incidencia.

3.4. Extensiones de las PN

Las extensiones de las PN surgen por la necesidad de modelar y cubrir las características de sistemas que no se manejan con evento-condición. En una PN ordinaria el peso de todos los arcos siempre es 1, solamente hay un tipo de token y la capacidad de un lugar para albergar tokens es

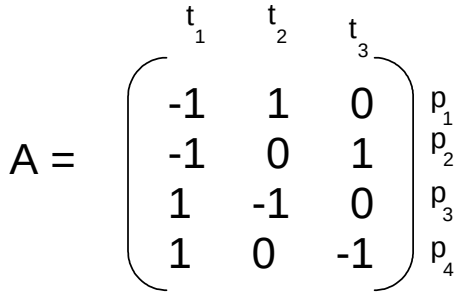

Figura 3.15: Matriz de incidencia de la red de Petri en la Figura 3.14.

infinita. En estas redes, una transición puede dispararse si cada lugar que la precede contiene al menos un token y el tiempo no se considera.

Las extensiones de PN están conformadas por modelos en los cuales se han agregado reglas de funcionalidad, para mejorar el modelo original. Dentro de las extensiones podemos considerar tres subclases importantes; la primera corresponde a modelos que tienen el poder de descripción de máquinas de Turing: PN con arco inhibidor y PN con prioridad. La segunda se relaciona con las extensiones que pueden modelar PN continuas y PN híbridas. La tercera atañe a las PN no-autónomas, las cuales describen el funcionamiento de aquellos sistemas cuya evolución es considerada por eventos externos y/o por tiempo: PN sincronizadas, PN con tiempo, PN interpretadas y PN estocásticas.

Mencionaremos algunas de las más interesantes y aquellas que nos sirvan para el desarrollo de esta tesis.

arco inhibidor.

El arco inhibidor es un arco dirigido que conecta un lugar p_i y una transición t_i. Este arco tiene un pequeño círculo al final del arco. En la Figura 3.16 se muestra un arco inhibidor que conecta a p_2 y t_4, significa que t_4 se podrá disparar solo si p_2 no contiene tokens. El disparo consiste en quitar un token de todos los lugares de entrada, excepto p_2, y aumentarle un token al lugar de salida.

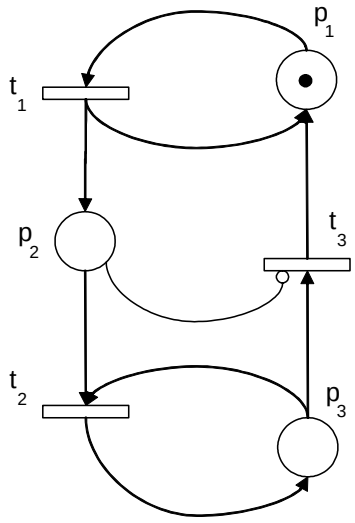

Figura 3.16: PN con arco inhibidor

Red de Petri Coloreada (CPN)

Este tipo de redes combina el poder de las redes de Petri con el de un lenguaje de programación. Las redes de Petri proveen las primitivas para la descripción de las transacciones concurrentes y el lenguaje de programación provee las primitivas para la definición de tipos de datos y la manipulación de estos [19][20].

La expresión en los arcos indica como cambiara el estado de la red y cuando las transacciones se ejecutaran. Cada token carga con un dato llamado color del token, el cual puede ser investigado y modificado cuando una transición se activa.

Las redes de Petri coloreadas tienen ese nombre porque permiten que los tokens carguen valores de datos y por supuestos que se puedan distinguir entre ellos, esto es los token poseen cierto "color", estos colores pueden ser manipulados y modificados cuando la transacción se activa. Para que una transición se active, necesita tener los suficientes tokens en los lugares de entrada, y los valores de los tokens deben ser igual a la expresión de los arcos.

Un ejemplo de CPN se muestra en la figura 3.17 , donde se describe un protocolo de transporte sencillo, el cual transfiere un número de paquetes de información a través de una red poco confiable

desde un origen hacia un destino. Cada lugar contiene un conjunto de marcas (tokens). Como se mencionó antes, cada uno de estos tokens lleva un dato, el cual pertenece a un tipo de dato. El lugar enviar (en la esquina superior izquierda de la figura 3.17) tiene siete tokens en su estado inicial. Todos los valores de los tokens pertenecen al tipo *INTxDATA* y representan siete paquetes que están listos para enviarse. El primer elemento en cada pareja de datos es el número del paquete y el segundo elemento es la información que se envía. Todos los 1's al frente de los apóstrofes indican que hay exactamente un token por cada uno de los paquetes definidos (en general, un lugar puede tener varios tokens con la misma información). Los lugares *SigEnvío* y *SigReg* inician con un solo token con valor 1 (que pertenece al tipo de dato *INT*). Estos lugares representan dos contadores, almacenando el número del siguiente paquete que será envíado/recibido. El lugar Recibido inicia con un token que contiene la cadena vacía "" (que pertenece al tipo de dato *DATA*). En este token se estarán concatenando las cadenas que se vayan recibiendo. Los lugares restantes *A*-*D* no tienen tokens en el estado inicial. Estos lugares representan los buffers de entrada y salida en el proceso de transmisión de la red [21].

3.5. Comentarios finales

Las PN poseen características intrínsecas para modelar y simular sistemas concurrentes. Además cuentan con herramientas matemáticas para su análisis. Estas propiedades son las bases por las cuales se eligieron las PN. El objetivo de esta tesis es poder en primera instancia modelar transacciones concurrentes en una base de datos. Después de la correcta interpretación gráfica es necesario analizarla. Muchos de los algoritmos existentes para detectar candados mortales en la bibliografía pretenden encontrar ciclos gráficamente. El análisis sobre una gráfica es más complicado que uno matricial. Este seria otro punto importante por el cual consideramos que las PN son la herramienta idónea.

En el siguiente capitulo ya entraremos a tratar el objetivo de esta tesis. Mezclando a las PN con los candados mortales y aquellas soluciones en las que concluimos

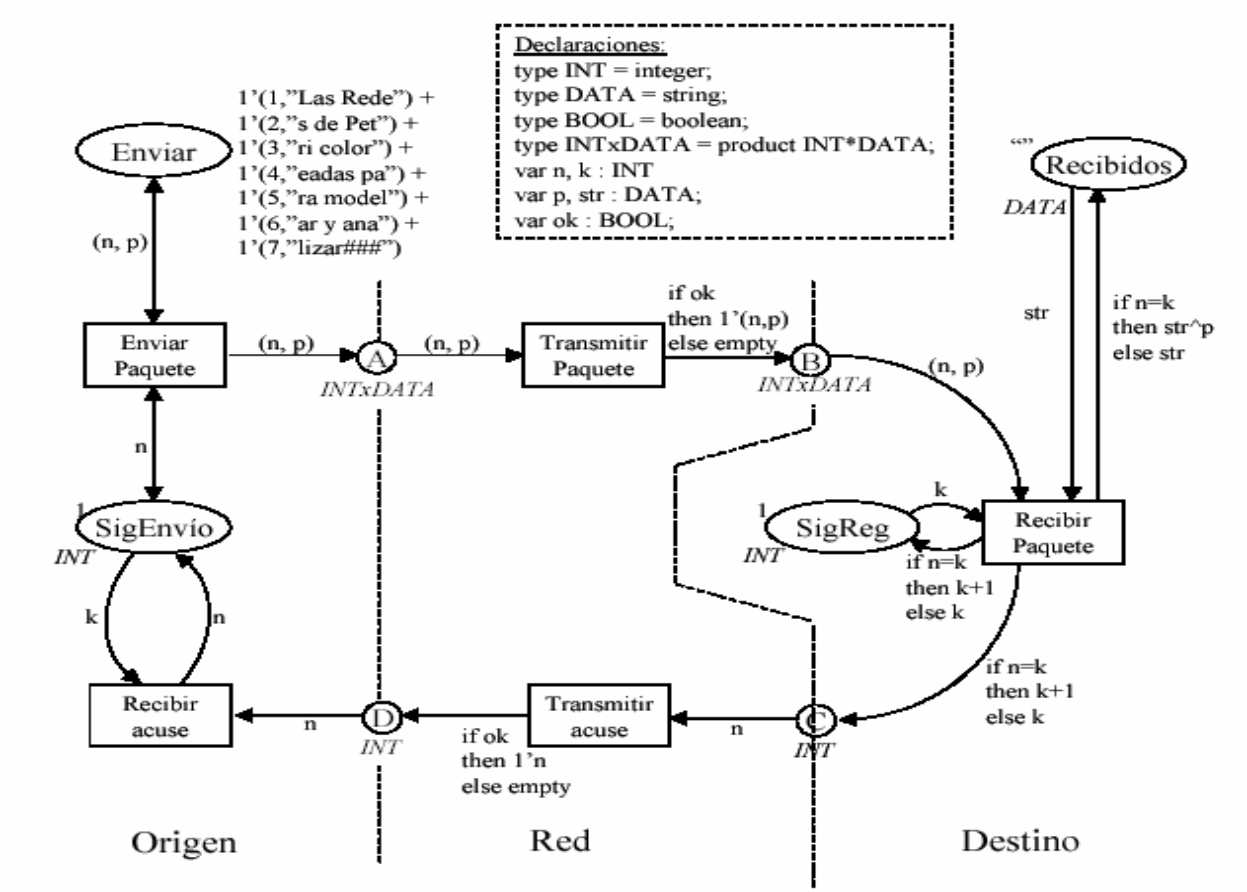

Figura 3.17: Una red de Petri coloreada

Capítulo 4

Análisis de candados mortales con PN

En el Capitulo 2 y 3 se plantean las bases teóricas necesarias para poder realizar un análisis de candados mortales con PN. Que es el objetivo de este capitulo. Primero mencionaremos que tipo de dependencias se manejaran para las transacciones y en base a estas modelaremos las PN. Mencionaremos todos los algoritmos involucrados para generar una herramienta que automatice la modelación y el análisis. Por ultimo comentaremos algunos ejemplos.

4.1. Sintaxis para modelar transacciones

Un conjunto de transacciones pueden tener dos tipos de dependencias: la dependencia de datos y de transacciones. La dependencia de transacciones (commit y abort) establece un orden de ejecución. La dependencia de datos (write y read), especifican la operación de lectura o escritura sobre un mismo registros en alguna base de datos.

La dependencia commit entre las transacciones T_i y T_j, se establece de la siguiente manera: $T_i \rightarrow T_j$. Implicando que T_i no podrá terminar su ejecución hasta que T_j finalice exitosamente. Por otro lado la dependencia abort: $T_i \longleftarrow T_j$. Específica que el aborto de T_j es condición suficiente para que T_i aborte.

La dependencia de datos ocurre cuando dos o más transacciones tratan de acceder a un mismo dato. Para asegurar que solo una transacción modifique el dato, se utiliza el control de concurrencia basado en candados (lock). Por ejemplo, si una transacción desea escribir en el dato x, tiene que

solicitar su candado. Si el candado esta disponible se le otorga y la transacción podrá acceder al dato x. En caso contrario tendrá que esperar por el candado.

Se analizo la dependencia de datos y transacciones, y se clasificaron de la siguiente manera:

$R1 < tdep >=< cdep >< adep >$

$R1{,}1 < cdep >= Ti \rightarrow < term >$

$R1{,}2 < adep >= Ti \longleftarrow < term >$

$R2 < ddep >=< wdep >< rdep >$

$R2{,}1 < wdep >= write\ nom\_trans\ dato$

$R2{,}2 < rdep >= read\ nom\_trans\ dato$

$R3 < term >= (< ANDterm >)$

$R4 < term >= (< ORterm >)$

$R5 < ANDterm >= (< ANDterm >)AND(< ANDterm >)$

$R6 < ORterm >= (< ORterm >)OR(< ORterm >)$

$R7 < AND - ORterm >= (< ADNterm >)OR(< ANDterm >)$

$R7{,}1 < AND - ORterm >= (< ORterm >)AND(< ORterm >)$

Dónde:

$< tdep >$ Son las dependencias de transacciones, dentro de las que se encuentran $< cdep >$ la dependencia commit y $< adep >$ la dependencia abort.

$< ddep >$ Son las dependencias de datos, dentro de las que se encuentran $< wdep >$ la dependencia de escritura y $< rdep >$ la dependencia de lectura.

De igual manera que en [1] existen los operadores lógicos AND y OR.

Ejemplo 4.1 *Una expresión correcta seria de la siguiente manera:*

$write\ T_i\ p$

$read\ T_j\ p$

$T_i \rightarrow (T_j\ AND\ T_r)$

El ejemplo indica que la transacción T_i necesita escribir en la tabla "p", la transacción T_j requiere hacer una operación de lectura sobre la misma tabla. Lo anterior indica la dependencia de datos que existen entre las transacciones. Por otro lado el orden de las ejecuciones se encuentra de la siguiente manera. T_i no puede ejecutar commit hasta que T_j y T_r terminen su ejecución satisfactoriamente. Es decir ejecuten commit.

La sintaxis aquí propuesta tiene como base el trabajo desarrollado por [1] en ese trabajo proponen una sintaxis para modelar un conjunto de transacciones con WFG.

La diferencia entre ambas sintaxis es que nosotros contemplamos la dependencia de datos para modelarlas. Sin embargo no es la única diferencia mas adelantes mencionaremos que mejoras se realizaron.

4.2. Modelación de transacciones con PN

La idea básica, es poder modelar un grupo de transacciones con las dependencias antes mencionadas, para esto se utilizan las redes de Petri. Pero es necesario tener en consideración los siguientes puntos.

- Cada transacción tiene asociada una transición.
- La dependencia de transacciones AND, tiene asignada una transición especial. La cual llamaremos t_{and}.
- La dependencia de transacciones OR, tiene asignado un lugar especial. Al cual llamaremos p_{or}.
- La dependencia de datos ($write\ T_i\ x$), se debe colocar un lugar del dato compartido "x". Al cual llamaremos p_{attach}.
- La dependencia de datos ($read\ T_i\ x$) se debe colocar un lugar del dato compartido "x"(al que nombramos p_{attach}), y el arco que conecta el lugar compartido con la transición debe ser inhibidor.
- En el commit $\rightarrow$ y el abort $\longleftarrow$, la declaración del lado derecho deben ser las precondiciones, para activar la transacción del lado izquierdo.

La forma de modelar estas dependencias se definirán de acuerdo a las dependencias establecidas previamente. La regla $R1$ indica que pueden existir dos tipos de dependencias *cdep* (dependencia commit) y *adep* (dependencia abort). Además esta regla genera $R1{,}1$ y $R1{,}2$. Las cuales especifican como se definen las dependencias commit y abort. Es importante resaltar los símbolos que las

representan $(\rightarrow, \longleftarrow)$ que corresponden al *commit* y al *abort* respectivamente. Por otro lado se componen de dos términos T_i y $< term >$. Donde $< term >$ es el predicado que se necesita cumplir para que la transacción T_i pueda ejecutar *commit* o *abort* dependiendo el caso.

La regla $R2$ indica las dependencias de datos que pueden existir *wdep* (escritura de un dato) y *rdep* (lectura de un dato). Esta regla genera dos más ($R21$ y $R22$), donde se define la sintaxis correcta para la lectura y escritura de datos. Están compuestos por una palabra reservada (*read* o *write*) el nombre de la transacción y el nombre del dato al que se le aplicará la operación de lectura o escritura.

De la regla $R3$ a la $R7{,}1$ se definen el predicado $< term >$. Los cuales corresponden a las posibles combinaciones de los operadores lógicos AND y OR.

Veamos mas a detalle las reglas definidas y comentemos como en base a estas se genera la red de Petri. Para la regla $R2$ y $R2{,}1$ una expresión correcta seria *write* T_i x. En esta se especifica que la transacción T_i desea escribir en la variable x. Recordemos que cuando una transacción desea leer algún dato no necesita candado. Por lo tanto, no necesita token en el lugar compartido; por este motivo se coloca un arco inhibidor. Si la transacción desea escribir, necesita un candado para poder acceder.

Para simularlo en primer lugar se define una transición que represente a la transacción T_i, el lugar de entrada a está será un lugar especial p_{attach} que simulara la variable. Por último un arco que los conecte. Esto se puede ver en la figura 4.1.

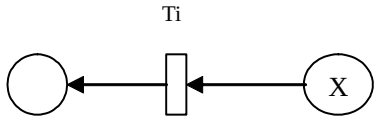

Figura 4.1: Dependencia de datos (*write* Ti x)

Para las reglas $R2$ y $R2{,}2$ una sentencia correcta seria T_i *read* x. La simulación es muy similar a la de escritura. De igual manera contamos con una transición que represente a T_i y un lugar especial p_{attach}. La diferencia existe en el arco, en esta regla debe ser un arco inhibidor. Puesto que la lectura no necesita candado. Este ejemplo se muestra en la Figura 4.2.

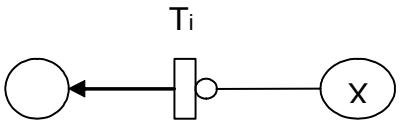

Figura 4.2: Dependencia de datos ($read\ Ti\ x$)

Si se desea simular la siguiente expresión ($T_i\ AND\ T_j$) de acuerdo a la regla $R3$. Necesitamos dos transiciones que representen a las transaccionesT_i y T_j; y una especial que realice la conjunción de estas dos t_{and}. Es vital recordar que T_i y T_j deben terminar exitosamente su ejecución por tal motivo sus lugares de salida deben ser la entrada de t_{and}. Para que t_{and} se ejecute necesita dos tokens los generados por T_i y T_j. Esto lo podemos ver en la Figura 4.3.

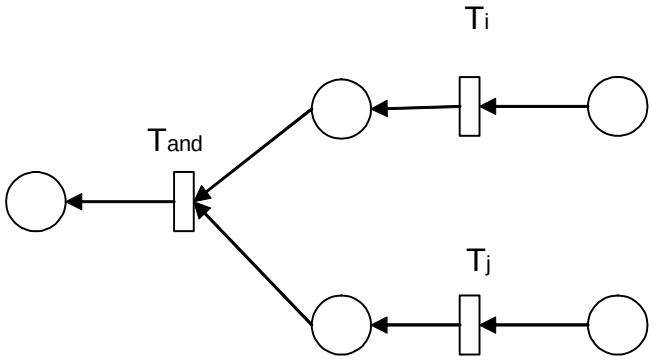

Figura 4.3: Dependencia de transacciones AND

En la regla $R4$ termino correcto seria ($T_i\ OR\ T_j$). Donde de igual manera se necesitan dos transiciones que representen a T_i y T_j. En esta caso la disyunción la representa un lugar especia p_{or}. Donde la ejecución satisfactoria de cualquiera de las dos transiciones colocaría un token a p_{or}. Este ejemplo lo podemos ver en la figura 4.4

Es importante mencionar que todas estas reglas se conjugan para generar ejemplos más complejos

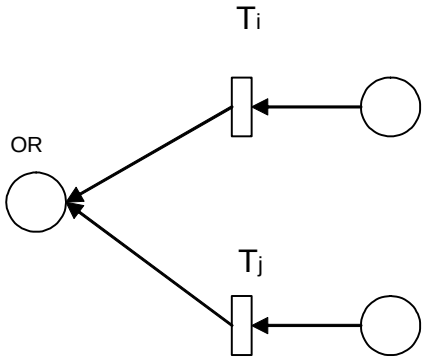

Figura 4.4: Dependencia de transacciones OR

4.3. Implementación del editor de redes de Petri.

El algoritmo trabaja de la siguiente manera. El primer paso es editar un grupo de transacciones con sus respectivas dependencias. Se analizan en base a las reglas de dependencia (ver sintaxis).

Por ejemplo si tenemos la siguiente dependencia: $T_i \rightarrow (T_j\ AND\ T_r)$. La composición commit y la expresión AND se encuentran especificadas por las reglas $R1{,}1$ y $R3$ respectivamente. Como podemos ver en el diagrama de flujo se realiza un ciclo para analizar todas las expresiones.

La expresión se divide en dos partes. La primera parte esta compuesta por T_i y la segunda parte por $T_j\ AND\ T_r$. La segunda parte es la precondición de la primera parte y es la que modelaremos primero.

En primer lugar necesitamos verificar si existe una transición que represente a las transacciones T_j y T_r, en caso de que no existan las creamos. Se verifica si tienen lugares de entrada, sino tienen, se agregan y se conectan con la transición, en caso contrario conectamos la transición con el lugar.

Para los lugares de salida hay dos posibilidades si expresión fuera OR se aumenta un lugar especial y las transiciones se conectan a esta, en el caso de la expresión AND se tienen que crear dos lugares de salida para cada transición. En ambos casos se conectan los arcos a sus respectivos lugares.

Para la expresión AND se requiere de una transición adicional, sus lugares de entrada serán los lugares de salida que componen la expresión, en este ejemplo T_j y T_r. Por lo tanto solo se requiere conectarlos. Se tendrá que crear un lugar de salida de la transición AND, la cual será el lugar de

entrada de la primera parte de la expresión (T_i).

Pasamos ahora a la primera parte de la expresión. Necesitamos una transición más que represente a T_i y conectar un arco del lugar de salida de la transición AND y este.

Este ejemplo lo podemos ver más claramente en el diagrama de flujo de la aplicación. La primera parte Figura 4.5 muestra la creación de las transiciones con sus correspondientes lugares de entrada y salida, así como sus arcos que los conectan.

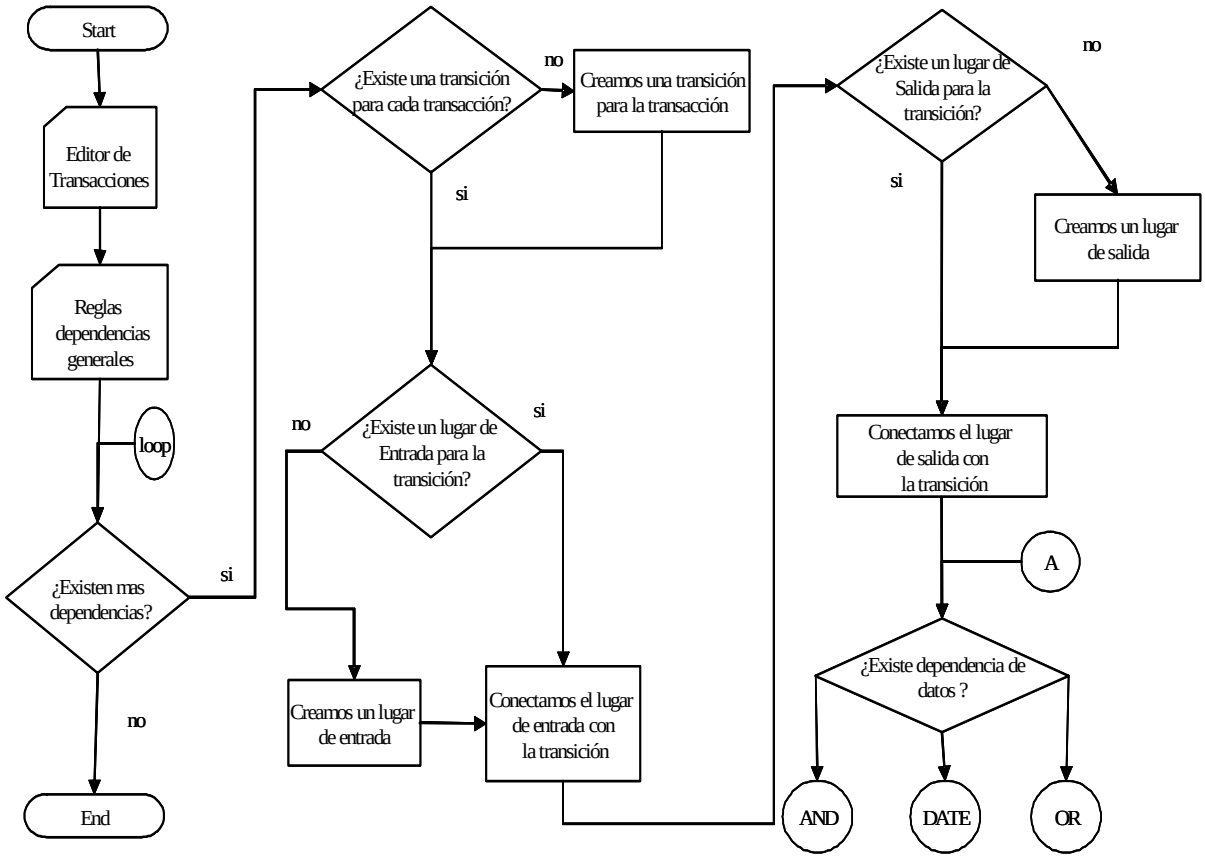

Figura 4.5: Diagrama de flujo de la generación de redes de Petri

En la Figura 4.6 muestra la creación de las expresiones AND y OR.

En la figura 4.7 se muestra la creación de las dependencias de datos *write* y *read*.

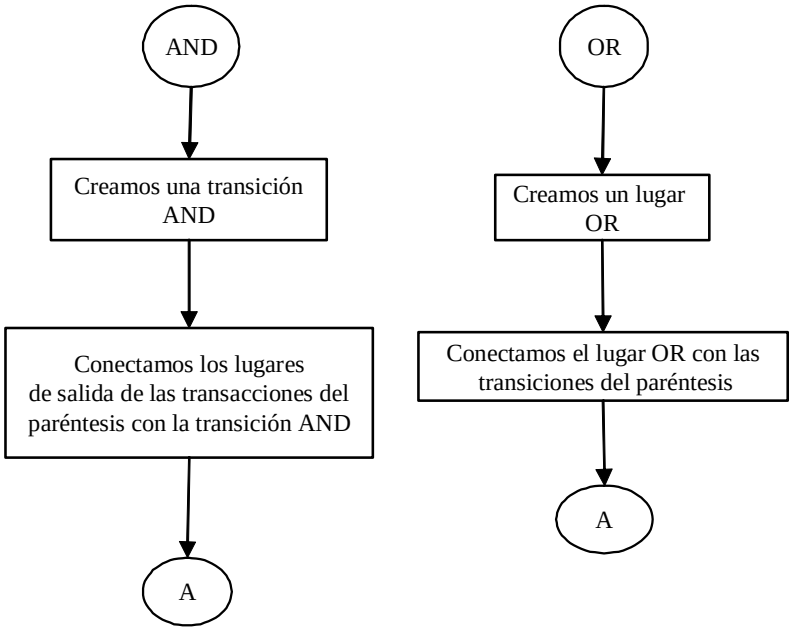

Figura 4.6: Diagrama de flujo de las dependencias AND y OR.

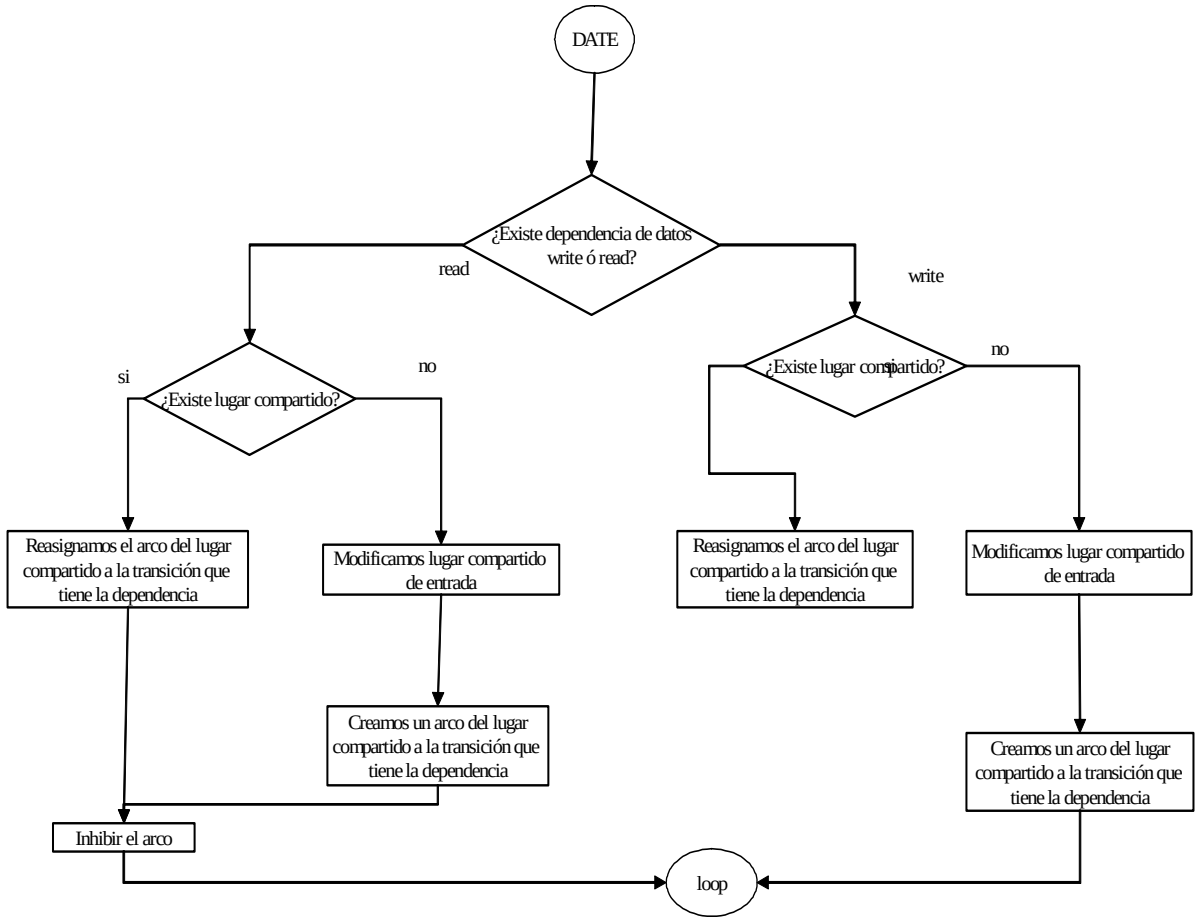

Figura 4.7: Diagrama de la dependencia de datos

4.3.1. Variables.

Las variables que se utilizan en la aplicación son básicamente arreglos o estructuras, donde almacenamos, en primer lugar el conjunto de transacciones que se desean modelar estas se almacenan en DG. La variable DG se analiza gramaticamente, extrayendo los símbolos que se encuentran en esta. Este análisis se realiza por medio de la sintaxis mencionada al inicio del capitulo.

Si el estudio gramatical termina incorrectamente no se podrá visualizar una PN . Por el contrario, si la gramática es correcta ahora analizaremos semánticamente el conjunto de transacciones.

Como vimos en el capitulo 3 una PN es una 4-tupla constituida por (P, T, I, O) donde P es el conjunto de lugares y T el con junto de transiciones que conformas a la PN. Los arcos que conectan a estos dos tipos de nodos se encuentran especificados en las estructuras $Ain(I)$ y $Aout(O)$.

Para entender el comportamiento de las estructuras Ain y $Aout$ pongamos un ejemplo, si tenemos:

$Ain(t_1) = \{p_1\}$

Nos indicara que p_1 es un lugar de entrada a la transición t_1. imaginemos que tenemos lo siguiente:

$Aout(t_1) = \{p_1\}$

Indica que p_1 es lugar de salida de t_1.

La variable Matriz contendrá la estructura de la PN, el tamaño de esta dependerá del numero de lugares (P) por el numero de transiciones (T). El estudio de esta matriz bidimensional se almacenara en rutas y será la variable que nos indique el comportamiento de las transiciones que se disparen.

DG. Es una estructura de datos que almacena las dependencias de transacciones y de datos que se desean modelar con redes de Petri.

P. Es el conjunto de lugares. Pueden ser lugares de entrada $({}^\circ p)$, lugares de salida (p°), lugares con un dato adjunto p_{attach}, y un lugar especial p_{or} que indica una dependencia OR.

T. Es el conjunto de transiciones. Existe una transición especial T_{and} que indica una dependencia AND.

Ain. Los arcos que conectan de un lugar a una transición. Existe un arco inhibidor A_{inh}

Aout. Los arcos que conectan de una transición a un lugar.

Matriz. Almacenamos la estructura de la PN para analizar candados mortales

Ruta. Variable donde almacenamos las rutas encontradas en la matriz de incidencia

4.3.2. Algoritmos.

Como entrada tendremos las dependencias generales, por ejemplo *cdep* significa que hay una dependencia *commit*. Como salida tendremos una red de Petri que modela el conjunto de transacciones de entrada.

INPUT: $DG = \{cdep, adep, wdep, rdep\}$

OUTPUT: Red de Petri del conjunto de dependencias generales

begin

$if\ not\ \exists t,\ [t = Transacción\ en\ DG, t \in\ T] \&\&\ \exists\ p[p \in P|^{\circ}p]$

/* Verificamos si existen transiciones y lugares asociados a estas. Si no existen tenemos que crearlos y conectarlos. */

$P = P \cup \{^{\circ}p\}$// Creamos el lugar de entrada

$t \longleftarrow\ TransacciónDG$ // Asignamos la transición con una transacción

$T \longleftarrow T \cup \{t\}$

// Creamos la transición de acuerdo a las transacciones del conjunto DG

$Ain \longleftarrow Ain \longleftarrow \{(p,t)\}$ //Conectamos el lugar de entrada con la transición

else

$Ain \longleftarrow Ain \cup \{(p,t)\}$

/*Solo conectamos los lugares existentes a la transición, cuando la transición y el lugar ya existen.*/

$if\ not\ \exists p\ [p \in P|p^{\circ}]$//Verificamos si no existe lugar de salida para la transición

$P = P \cup \{p^{\circ}\}$ // Creamos el lugar de salida

$Aout \longleftarrow Aout \cup \{(t,p)\}$ //Conectamos la transición con el lugar de salida

else

$Aout \longleftarrow Aout \cup \{(t,p)\}$ //Conectamos el arco

// En este bloque ya modelamos el conjunto de transacciones de entrada

end

/* Este bloque modela las dependencias de transacciones AND */

$if\ term == AND$

begin

$t_{and} \longleftarrow T_{AND}$ //Asignamos la nueva transición con la dependencia AND

$T \longleftarrow T \cup \{t_{and}\}$ // Creamos la transición AND

$Ain \longleftarrow Ain\{(p^{\circ}, t_{and})\}$

/*Conectamos los lugares correspondientes con la nueva transición.*/

end

/* Este bloque modela las dependencias de transacciones OR */

$if\ term == OR$

begin

$p_{or} \longleftarrow P_{OR}$//Asignamos la nueva transición con la dependencia OR

$P = P \cup \{p_{or}\}$ //Creamos el lugar OR

$Ain \longleftarrow Ain\{(t, p_{or})\}$ // Conectamos las transiciones al nuevo lugar OR

end

/* Este bloque modela la dependencia de datos write, para cuando una transacción requiere actualizar una tabla */

$if\ term == depw$

begin

//Verificamos si no existe un lugar con un dato adjunto

$if\ not\exists\ p\ [p \in P | p_{attach}]$

$P = P \cup \{^{\circ}p_{attach}\}$ // Modificamos el lugar de entrada con dato adjunto

$Ain \longleftarrow Ain\{(p_{attach}, t)\}$

else

$Ain \longleftarrow Ain\{(p_{attach}, t)\}$ //Reasignamos el arco

end

/* Este bloque modela la dependencia de datos read, para cuando una transacción requiere consultar una tabla */

$if\ term == depr$

begin

//Verificamos si no existe un lugar con un dato adjunto

$if\ not\ \exists p\ [p \in P | p_{attach}]$

$P = P \cup \{^{\circ}p_{attach}\}$ // Modificamos el lugar de entrada con dato adjunto

$Ain \longleftarrow Ainh\{(p_{attach}, t)\}$ //Conectamos el arco inhibidor

$else$

$Ain \longleftarrow Ainh\{(p_{attach}, t)\}$ //Reasignamos el arco inhibidor

end

4.4. Detección de candados mortales con Matriz de incidencia

La mayoría de los algoritmos de la literatura basan las detecciones de candados mortales buscando ciclos, algunos lo realizan por medio de gráficas (WFG), por medio de mensajes de prueba o con un historial de los procesos activos del sistema. Una solución alternativa son las redes de Petri. Algunas de las investigaciones realizadas al respecto y que se apegan más a nuestro tema de investigación son las efectuadas por Bertino, Chiola y Manzini. En esta sección proponemos un método diferente al propuesto en [1]. La matriz de incidencia posee la información necesaria de la estructura de una PN. El manejo de esta nos ofrece fundamentos necesarios para predecir el comportamiento de una transacción. Es decir, conocer a priori si las transacciones no tendrán conflicto alguno para su ejecución.

4.4.1. Análisis de la estructura de una PN con matriz de incidencia

La matriz de incidencia contiene información sobre la estructura de la red de Petri. El objetivo es analizar la estructura de la gráfica para la detección de candados mortales.

En las dependencias de transacciones la forma de detectar conflictos, es la detección de rutas cíclicas. Es decir que inicien y terminen en el mismo punto. Pero con la dependencia de datos esta técnica no funciona. Para la detección en dependencia de datos, se necesitan encontrar varias rutas que tengan como característica el mismo origen y el mismo destino.

Ejemplo 4.2 *Supongamos una PN que se muestra en la figura 4.8 donde la dependencia seria* $T_x \longleftarrow (T_i\ OR\ T_j)$. *La dependencia especifica que* T_x *no podrá ejecutar commit hasta que* T_i *o* T_j *terminen satisfactoriamente. La modelación de esta dependencia se muestra en la figura 4.8.*

Imaginemos que el lugar de T_i tienen un token, entonces tendría las precondiciones para habilitarse, al dispararse colocaría un token en el lugar T_{OR} y T_x se podrá habilitar. Esta seria una de las rutas posibles, se muestra con una línea punteada la ruta que seguiría.

Por otro lado si T_j se dispara esto generaría un token en T_{OR} y por ende T_x tendría sus precondiciones para dispararse. Las dos rutas posibles serán la línea punteada y la línea normal. Como podemos ver existen dos caminos para que T_x pueda terminar satisfactoriamente.

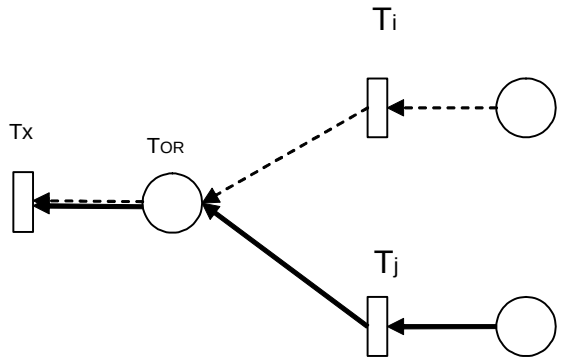

Figura 4.8: Búsqueda de rutas en la estructura de una PN

En la figura 4.9 se muestra la matriz de incidencia de la PN de la figura 4.8. Donde P_0 representa al lugar de entrada de T_i, P_1 es el lugar de entrada de T_j y P_2 es T_{OR}. Por otra parte t_0 es T_i, t_1 es T_j y t_3 es T_x. (para el llenado de la matriz ver cap. 3).

Para analizar las rutas (como mencionamos en el capitulo 3), podemos utilizar la matriz de incidencia. Para formar la ruta, primero se localiza en forma horizontal un nodo inicial con valor de –1. Posteriormente en la misma columna se localiza el valor de 1, siendo este el segundo nodo de la ruta. Después se busca en la fila el valor de –1 y el siguiente 1 se buscara en la misma columna y así sucesivamente. La figura 4.10 muestra una de las rutas de la PN.

La matriz de incidencia nos indica lo siguiente: la transición t_0 tiene como lugar de entrada a p_0 y tiene como lugar de salida a p_2 y que a su vez es la entrada a t_2. Como podemos ver esta ruta corresponde a la línea punteada de la figura 4.10. La siguiente ruta a encontrar se muestra en la figura 4.11.

Las rutas encontradas se forman por las coordenadas de la matriz de incidencia. Las rutas

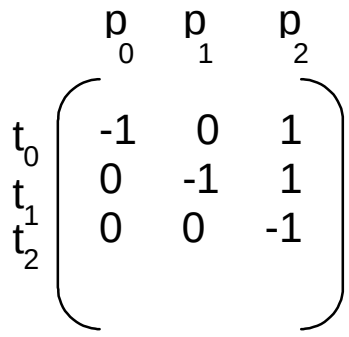

Figura 4.9: matriz de incidencia generada de la PN de la Figura 4.8

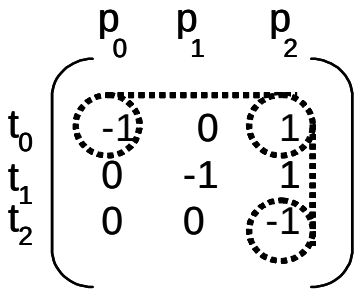

Figura 4.10: Ruta encontrada en la matriz de incidencia.

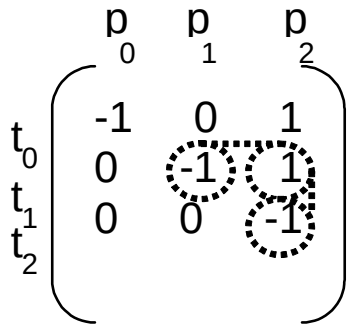

Figura 4.11: Ultima ruta encontrada en la matriz de incidencia.

quedarían de la siguiente manera:

Rutas/Valor	-1	1	-1
$R1$	$(0,0)$	$(0,2)$	$(2,2)$
$R2$	$(1,1)$	$(1,2)$	$(2,2)$

La conclusión seria que existen dos caminos diferentes para llegar al mismo destino los cuales son mutuamente excluyente.

Ya encontradas todas las rutas de la red, solo falta analizar estas para detectar candados mortales. Como mencionamos anteriormente se intenta detectar ciclos en la estructura. Por lo tanto si existen dos coordenadas iguales dentro de la ruta, querrá decir que existe un candado mortal. Este ejemplo esta libre de candados mortales.

Algoritmo 4.1 *El algoritmo que busca las rutas de una matriz de incidencia es el siguiente:*

```
int busqueda (int y1, int x1){
    x=x1;
    for ( y=y1;y<num_filas; y++ ){
        if (matrix[y][x]==-1){//Buscamos -1 en las columnas
            Imprime coordenada
    for( x=x1; x<num_columnas ;x++ ){
        if (matrix[y][x]==1){
            Imprime coordenada
                return busqueda (y,x);
        //Encontramos el primer par ordenado lo retornamos y buscamos el siguiente con estas
coordenadas de inicio
                    }
                }
            }
        }
return 0; //si no hay mas coordenadas retornamos 0
}
```

El algoritmo retorna un valor de tipo entero que equivale a la coordenada encontrada. Consta de dos argumentos que corresponden a las coordenadas donde se iniciará la búsqueda (int y1, int x1); de acuerdo con este punto de inicio, se inicia la búsqueda de -1 en forma vertical. Al encontrarse esta coordenada ahora buscará 1 en forma horizontal. Al encontrar ambas coordenadas retornara la coordenada final. Iniciándose nuevamente la búsqueda. El algoritmo termina el recorrer toda la matriz.

Definición 4.1 *Una Ruta Cíclica RC es una ruta R donde el último par ordenado* (x, y) *ya se encuentra enlistado en R.*

Definición 4.2 *Una Ruta Acíclica RA es aquella donde la última pareja ordenada* (x, y) *es diferente a sus antecesoras.*

De estas definiciones presentamos el siguiente teorema para la detección de candados mortales:

Teorema 4.1 *Si existe una RC dentro del conjunto de rutas R de una PN, entonces existe un candado mortal en la dependencia de transacciones.*

Teorema 4.2 *Si existe mas de dos RA dentro del conjunto de rutas de una PN que tengan las mismas coordenadas de inicio y final, entonces existe un candado mortal en la dependencia de datos.*

Demostración. Si existe una transición $t_2 \in T$ tal que su ejecución sea la precondición para el disparo de la transición $t_0 \in T$. Que a su vez es la precondición t_2 nos encontramos con un ciclo y como consecuencia tendríamos un disparo infinito de las transiciones. ■

Demostración. Si existe un lugar $p_0 \in P$ tal que sea el nodo inicial de dos rutas diferentes $R1$ y $R2$; y que la transición $t_1$sea el nodo terminal de estas dos rutas. Si esto ocurriese tendríamos un problema de acceso a un dato compartido. ■

4.5. Ejemplos Prácticos

Ejemplo 4.3 *Para la dependencia:*

$T_i \rightarrow T_j$

$T_j \rightarrow (T_x \; OR \; T_i)$

Nos indica que T_i no podrá ejecutar commit hasta que T_j termine satisfactoriamente. Por otro lado T_j requiere que T_i o T_x ejecuten commit. Intuitivamente podemos ver que T_i necesita que T_j terminé y a su vez T_j tiene como precondición a T_i. Gráficamente se muestra en la Figura 4.12.

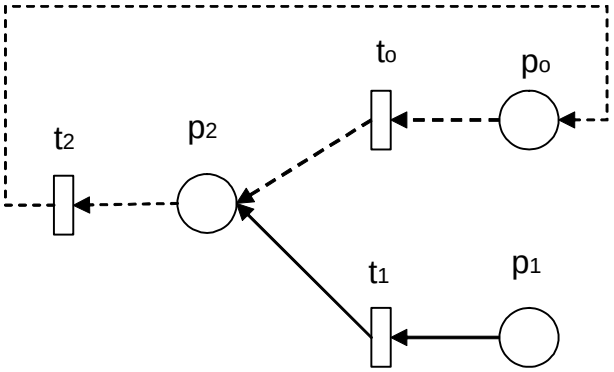

Figura 4.12: Una ruta cíclica en una PN con dependencia OR

La matriz de incidencia de la Figura 4.12 seria la que se muestra en la siguiente tabla:

$$\begin{bmatrix} & p_0 & p_1 & p_2 \\ t_0 & -1 & 0 & 1 \\ t_1 & 0 & -1 & 1 \\ t_2 & 1 & 0 & -1 \end{bmatrix}$$

En la Figura 4.12 se muestra una ruta cíclica formada por las líneas punteadas. Ahora analicemos las posibles rutas generadas a partir de la matriz de incidencia:

Rutas/Valor	-1	1	-1	1	-1
$R1$	$(0,0)$	$(0,2)$	$(2,2)$	$(2,0)$	$(0,0)$
$R2$	$(1,1)$	$(1,2)$	$(2,2)$	$(2,0)$	$(0,0)$

Como podemos ver la $R1$ inicia y termina en la misma coordenada la $(0,0)$, gráficamente podemos ver el mismo resultado. Si el p_0 tuviera un token habilitaría t_0 que a su vez habilitaría

a t_2. El problema resulta en que t_2 colocaría un token para que t_0 se habilite. Entonces caeríamos en un disparo infinito de las transiciones t_0 y t_2. Por otra parte si la transición t_1 se disparara colocaría un token para que t_2 se dispare y este a su vez a t_0. Como podemos ver, la ejecución de estas transacciones nos traería como resultado un estado inconsistente de la base de datos.

Ejemplo 4.4 *Imaginemos que le cambiamos la dependencia para que quede de la siguiente manera*

$T_i \rightarrow T_j$

$T_j \rightarrow (T_x\ AND\ T_i)$

La PN generada se muestra en la Figura 4.13. De igual manera presenta un ciclo el cual podemos ver con las líneas punteadas.

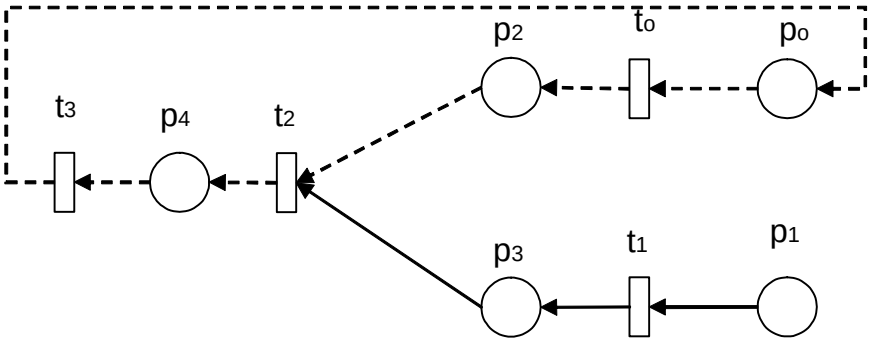

Figura 4.13: Una ruta cíclica en una PN

La matriz de incidencia de la Figura 4.13 seria la que se muestra a continuación:

$$\begin{bmatrix} & p_0 & p_1 & p_2 & p_3 & p_4 \\ t_0 & -1 & 0 & 1 & 0 & 0 \\ t_1 & 0 & -1 & 0 & 1 & 0 \\ t_2 & 0 & 0 & -1 & -1 & 1 \\ t_3 & 1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

Ahora analicemos las posibles rutas generadas a partir de la matriz de incidencia:

Rutas/Valor	-1	1	-1	1	-1	1	-1
$R1$	$(0,0)$	$(0,2)$	$(2,2)$	$(2,4)$	$(3,4)$	$(3,0)$	$(0,0)$
$R2$	$(1,1)$	$(1,3)$	$(2,3)$	$(2,4)$	$(3,4)$	$(3,0)$	$(0,0)$

Como podemos ver la ruta $R1$ inicia y termina en la coordenada $(0,0)$. Por lo tanto nos encontramos con una ruta cíclica. Analicemos el comportamiento dinámico de la PN: imaginemos que p_0 tiene un token por lo tanto se habilita t_0 y coloca un token en p_2. En este momento t_2 no se podría ejecutar puesto que necesita que p_3 tenga un token, t_2 tendrá que esperar. Si t_1 se disparara, existirán las precondiciones para que t_2 se habilite. Al disparase t_2 se habilitaría t_3. En este caso todas las transiciones se ejecutaron correctamente, pero la transición t_0 se volvería a ejecutar. Sin embargo la ejecución de t_2 depende del disparo de dos transiciones. En este ejemplo no sucede un disparo consecutivo y por ende el estado del sistema se comporta más estable.

Los ejemplos anteriores son para la dependencia de transacciones. Para la dependencia de datos las rutas cíclicas no funcionan para detectar conflictos en la base de datos. Esto es, porque cuando varias transacciones acceden al mismo dato pueden ocurrir varios casos. Por ejemplo, si T_i escribe y T_j lee en x, puede pasar lo siguiente: T_i toma el candado de x, entonces, T_j tendrá que esperar hasta que T_i libere el recurso. Podría darse un candado mortal. Pero si T_j lee x no hay posibilidad de conflicto. Es por estas circunstancias variables no se puede asegurar que exista un candado mortal. Por lo tanto solo mencionaremos que es probable que ocurra.

Para tratar el problema antes mencionado, necesitamos encontrar varias rutas que inicien en el mismo lugar. Esto es, tenemos que detectar dos o más valores de -1 en las columnas. Como sabemos las columnas representan los lugares en la red de Petri y el valor de -1 indica que es un lugar de entrada a una transición. Por lo tanto, un mismo lugar sale hacia dos rutas diferentes. Solo faltaría comparar las coordenadas finales de la ruta para saber que inician y terminan en el mismo lugar.

Ejemplo 4.5 *Veamos el siguiente ejemplo.*

$write\ T_i\ x$
$read\ T_j\ x$
$T_i \rightarrow (T_j\ AND\ T_r)$
La PN de la dependencia anterior se puede ver en la Figura 4.14.

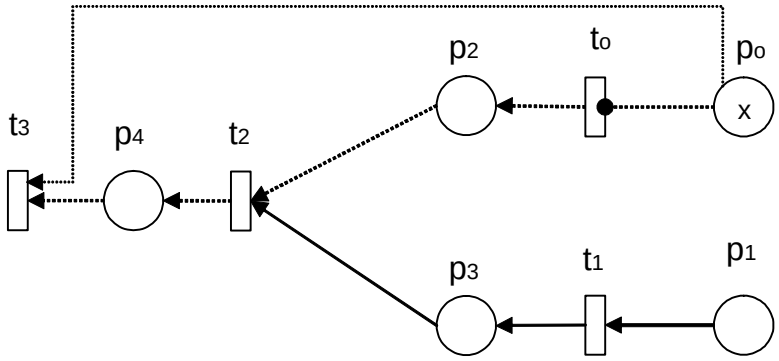

Figura 4.14: Una PN con dependencia de datos

En este ejemplo dos transacciones acceden a un dato compartido. Donde T_i escribe y T_j lee en "x". Por otra parte se especifica la siguiente dependencia T_i no podrá ejecutar commit hasta que T_j y T_r terminen su ejecución. La estructura de la gráfica es similar a los ejemplos anteriores. Cabe mencionar que el lugar compartido es p_0, el arco que conecta a p_0 con t_3 indica que esta transición requiere escribir en el lugar "x". Por otro lado, el arco que conecta a t_0 con p_0 es inhibidor, indica que t_0 no necesita de un token en p_0 para habilitarse y poder leer el dato "x".

Como se puede ver en la Figura 4.14 existen dos rutas (las líneas punteadas) que comparten el lugar de inicio. Además que son las precondiciones para habilitar t_3. Es altamente probable que exista un candado mortal. Imaginemos que el lugar p_0 tiene un token, t_0 no se habilitaría, pero una de las precondiciones de la transición t_3 estaría disponible. Sin embargo quedaría bloqueada

Ahora veamos otro caso, si p_0 no tuviera token la transición que se habilitaría seria t_0. Para que se dispare t_2 solo se tendrá que esperar la ejecución de t_1. Pensemos que en el instante de que dispare t_2 se coloca un token a p_0, con lo que se cumplirían las dos precondiciones para que la transición t_3 se habilite. En este caso no existiría candado mortal puesto que todas las transiciones se disparan correctamente. Como sabemos la ejecución de una transacción (en PN su similar son las transiciones) depende directamente de cuando el Sistema Operativo le asigne tiempo de procesador. Por lo tanto hay varias combinaciones para la ejecución de las transacciones (anteriormente mencionamos solo dos), esto hace que no podamos garantizar al cien por ciento que exista un candado mortal. Sin embargo podemos mencionar sus posibilidades.

La matriz de incidencia de la Figura 4.14 es:

$$\begin{bmatrix} & p_0 & p_1 & p_2 & p_3 & p_4 \\ t_0 & -1 & 0 & 1 & 0 & 0 \\ t_1 & 0 & -1 & 0 & 1 & 0 \\ t_2 & 0 & 0 & -1 & -1 & 1 \\ t_3 & -1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

Las rutas generadas serían:

Rutas/Valor	-1	1	-1	1	-1
$R1$	$(0,0)$	$(0,2)$	$(2,2)$	$(2,4)$	$(3,4)$
$R2$	$(1,1)$	$(1,3)$	$(2,3)$	$(2,4)$	$(3,4)$
$R3$	$(2,2)$	$(2,4)$	$(3,4)$		
$R4$	$(2,3)$	$(2,4)$	$(3,4)$		
$R5$	$(3,0)$				

Estas serían todas las rutas que se generarían partir de la matriz. Sin embargo no todas nos proporcionan información para el análisis, por ejemplo la ruta $R3$ es la parte final de la ruta $R1$. Lo mismo sucede con las rutas $R2$ y $R4$. Obviamente las rutas $R3$ y $R4$ no se analizan. De esta manera nuestro espacio de búsqueda se recorta brindando mas eficiencia al algoritmo.

4.6. Comentarios finales

La detección de candados mortales es un tema muy tratado. Muchos de los algoritmos existen ya están muy probados. Sin embargo las redes de Petri le dan una visión diferente a este tema. Las herramientas con la que cuenta facilitan desde su modelación (gráficos más sencillos y entendibles) así como herramientas para su análisis. Una de estas fue la que se utilizo en esta tesis.

La matriz de incidencia nos permite analizar la estructura de la PN. Para fines prácticos resulta más sencillo un estudio sobre una matriz que uno realizado a una gráfica. Sin embargo, la matriz puede crecer demasiado. Esto puede ser una dificultad. La solución a esto resulta muy sencilla. No es necesario generar un gran número de rutas puesto que no todas nos brindan la información que necesitamos. Esto es porque solo nos interesan los primeros nodos de la red. Por ejemplo para la

figura 4.14 nos importan el disparo de las transiciones t_0 y t_1. Con esto reducimos a un 50 % la búsqueda de rutas.

Capítulo 5

Plataforma de desarrollo

Existen una gran variedad de aplicaciones desarrolladas para modelar Redes de Petri dentro de las que se encuentran: redes de Petri estocásticos, coloreadas, de tiempo, etc. Para fines de esta tesis las que mas nos interesaron son aquellas que analizan la estructura de la red. Dentro de las que se encuentran PIPE (Platform Independent Petri Net Editor), JARP, Jfern (Java-based Petri Net framework), Petri Net Kernel entro otros (para mas información en [9]). No es objetivo de la tesis abundar en estas herramientas. Sin embargo contienen ciertas similitudes con la que se desarrollo. Por ejemplo PIPE proporciona una herramienta de diseño para editar la red de Petri y dentro de sus características mas interesantes para nuestro estudio cuenta con un análisis matricial, el cual consiste en mostrar la matriz de incidencia a priori y a posteriori de acuerdo a una marca inicial.

En la sección 5.1 se explica el desarrollo de TRANSimul, mencionando el lenguaje de programación que se utilizo y la arquitectura de su constitución. En la sección 5.2 se describe el diagrama de clases, así como una descripción de estas y sus métodos. Por último en la sección 5.3 se detalla el uso de la herramienta.

5.1. Planteamiento

Los candados mortales generan en la mayoría de los casos problemas con la inconsistencia de cualquier sistema. Es por este motivo el surgimiento de un sin fin de algoritmos para la prevención y la detección de este tipo de problemas. En esta tesis proponemos una herramienta que analiza un

conjunto de transacciones con dependencias. La herramienta que se utiliza para este objetivo son redes de Petri.

La aplicación ofrece un medio gráfico y visual para editar dependencia de datos y de transacciones, auxiliándose con la teoría de redes de Petri para su modelación y análisis.

De acuerdo al análisis se mencionará que probabilidades existen de la finalización exitosa de las transacciones.

5.2. Arquitectura

TRANSimul funciona de la siguiente manera, existe un TextArea y es donde se almacenan las expresiones por modelar. Para descomponer esta expresión en palabras individuales o unidades lexicográficas, se utiliza la clase StringTokenizer (divisor de Strings en unidades lexicográficas) de Java (del paquete java.util). Para utilizar está clase se crea un objeto StringTokenizer, y utilizamos los métodos nextToken() para extraer las unidades lexicográficas y el método hasMoreTokens() para saber cuando termina la expresión.

Ya analizada la expresión (de acuerdo a la gramática establecida), pasamos al siguiente paso, la interpretación de la expresión. El análisis nos lleva a la generación de la matriz de incidencia y en base a esta construimos la red de Petri. Por ultimo, buscamos todas las rutas posibles dentro de la matriz y detectamos los posibles candados mortales. Ver Figura 5.1.

En la Figura 5.4 se muestra la arquitectura de TRANSimul. Recordemos que la aplicación cuenta con una plataforma para poder interactuar con los usuarios. El bloque de Editor de transacciones es el que contiene el ambiente gráfico que se conecta con el usuario para ser el medio por el cual se edita el conjunto de transacciones a modelar. A su vez se conecta con el análisis gramatical que es el encargado de respetar la sintaxis aquí propuesta. Si la gramática es la correcta el bloque de análisis semántico es el encargado de interpretar lo que el usuario edito. En base a este estudio se comienza formar la Matriz de incidencia. Este bloque se divide en dos directrices por un lado se genera gráficamente la red de Petri. Por otro lado, el bloque de búsqueda de rutas analiza la matriz de incidencia y detecta todas las posibles rutas. El último bloque analiza las rutas arrojadas para la detección de candados mortales.

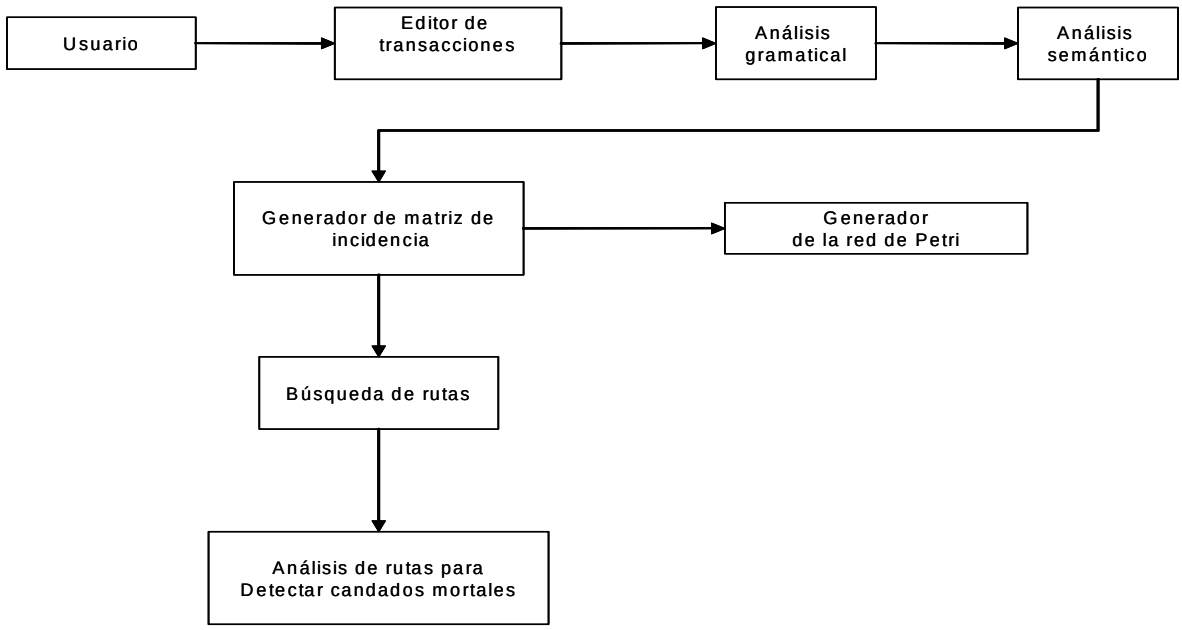

Figura 5.1: Arquitectura de TRANSimul

5.3. Diseño

El simulador le permite al usuario modelar un grupo de transacciones de acuerdo a la sintaxis mostrada en el capitulo 4. Para poder utilizar está aplicación debemos ejecutar la siguiente línea en la consola:

c:\> java Editor

Después de ejecutar la instrucción aparecerá la Figura 5.2.

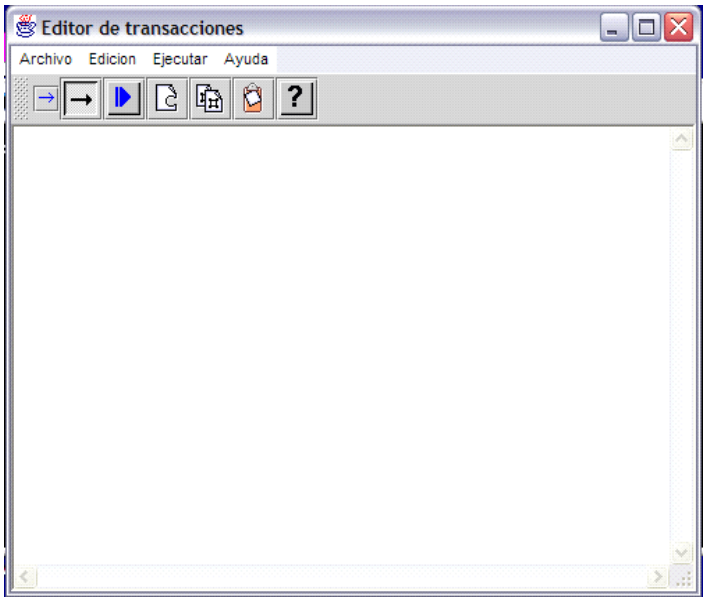

Figura 5.2: Editor de Transacciones.

La interface gráfica se desarrollo en el lenguaje de Programación Orientado a Objetos Java. Se optó por este lenguaje debido a que podemos tratar como objetos todos los componentes de la red de Petri (lugares, transiciones, arcos, etc.). Además para el análisis sintáctico es mas sencillo descomponer el texto en tokens y tratarlos como objetos facilitándonos el análisis semántico.

El editor de transacciones (TRANSimul) brinda la facilidad para editar un conjunto de transacciones de acuerdo a la sintaxis proporcionada en el capitulo 4. Cuenta con un analizador semántico para generar una red de Petri que corresponderá a la gráfica que modela el comportamiento de la ejecución de las transacciones concurrentemente. Además muestra la matriz de incidencia y realiza

un análisis estructural detectando aquellas transacciones que se encuentran en candado mortal. Para esto utiliza el algoritmo de la Sección 4.4

TRANSimul esta compuesto por un menú desplegable, una barra de herramienta y un área de edición. La cual se muestra en la Figura 5.3 . En el área de edición se capturan las dependencias de transacciones y de datos que se deseen modelar.

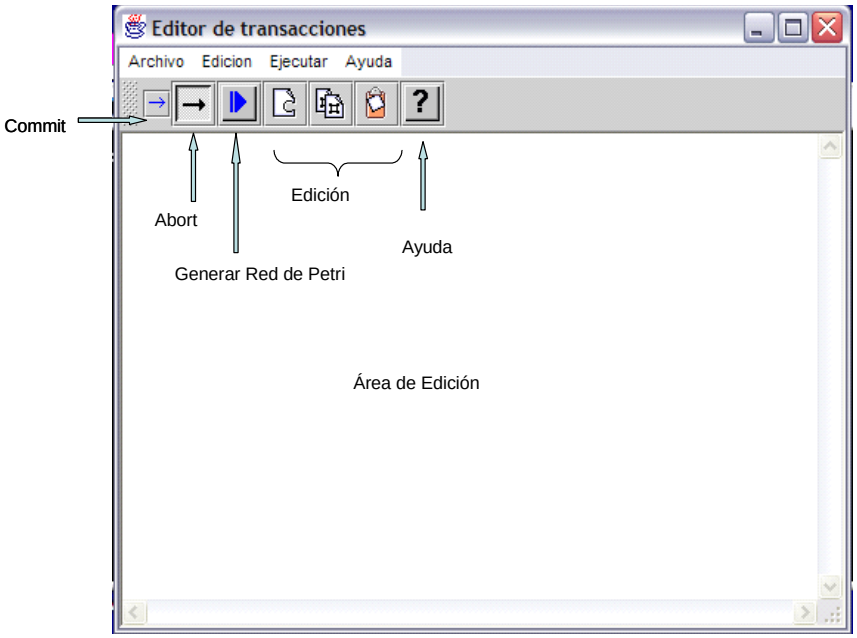

Figura 5.3: Editor de transacciones (TRANSimul)

La barra de tareas y el menú cuenta con elementos para ayudar a la edición. Por ejemplo las opciones de copiar cortar y pegar para manipular texto. Además un icono para generar la red de Petri cuando se tenga un grupo de transacciones editadas.

La barra de herramientas tiene tres agregados. Dos de ellos permiten manejar los símbolos de commit y abort; y por ultimo el de ayuda

La forma de utilizar el TRANSimul la explicaremos mas adelante.

5.4. Las clases.

La Programación Orientada a Objetos (POO) utiliza dos conceptos la herencia y la composición. Donde la herencia es el mecanismo por la cual se crean nuevos objetos en términos de objetos ya existentes. Es decir, para representar una dependencia de datos AND (ver Figura 4.3) se necesitan crear varios objetos de tipo lugar, transición y arco los cuales heredan los atributos de circulo, rectángulo y línea respectivamente.

La composición significa utilizar objetos dentro de otros objetos por ejemplo la dependencia AND es un objeto que contiene en su interior otros objetos como lugares, transiciones, arcos, etc.

Gráficamente la herencia se representa colocando la clase hija por debajo de la super clase y una flecha que los conecta. Por otro lado la composición se representa con una flecha que parte de la clase atributo hacia el lado izquierdo de la clase que contiene. El diagrama de clases se muestra en la figura5.4.

Las clases principales son Editor de texto, Red de Petri y Figuras. Donde Editor de texto es una extensión de Frame de java y es una composición de las clases Analizador , Menu y JToolBar. La clase Analizador esta compuesta de las cuatro dependencias de transacciones que modelamos (AND, OR, write y read), esta clase a su vez genera la red de Petri. Las clases de la dependencia AND, OR, write y read esta compuesta de la clase Figuras. Dentro de Figuras tenemos la transición, lugar y arco. Donde la clase transición hereda los atributos de la clase rectángulo que a su vez hereda de la clase punto. Algo similar sucede con las clases arco y lugar.

5.4.1. La clase Editor

Esta es la clase principal de la aplicación. Hereda los atributos y métodos de la clase Frame de Java. Proporciona una interface gráfica donde el usuario interactúa con el sistema para editar un conjunto de transacciones. Se implementa un área de texto para esté fin. Aquí se diseña una barra de herramientas que proporciona métodos de ayuda para la edición como copiar, pegar y cortar. Además cuenta con dos íconos para poder insertar el símbolo de commit y abort. Por último se implemento un icono para poder generar la red de Petri.

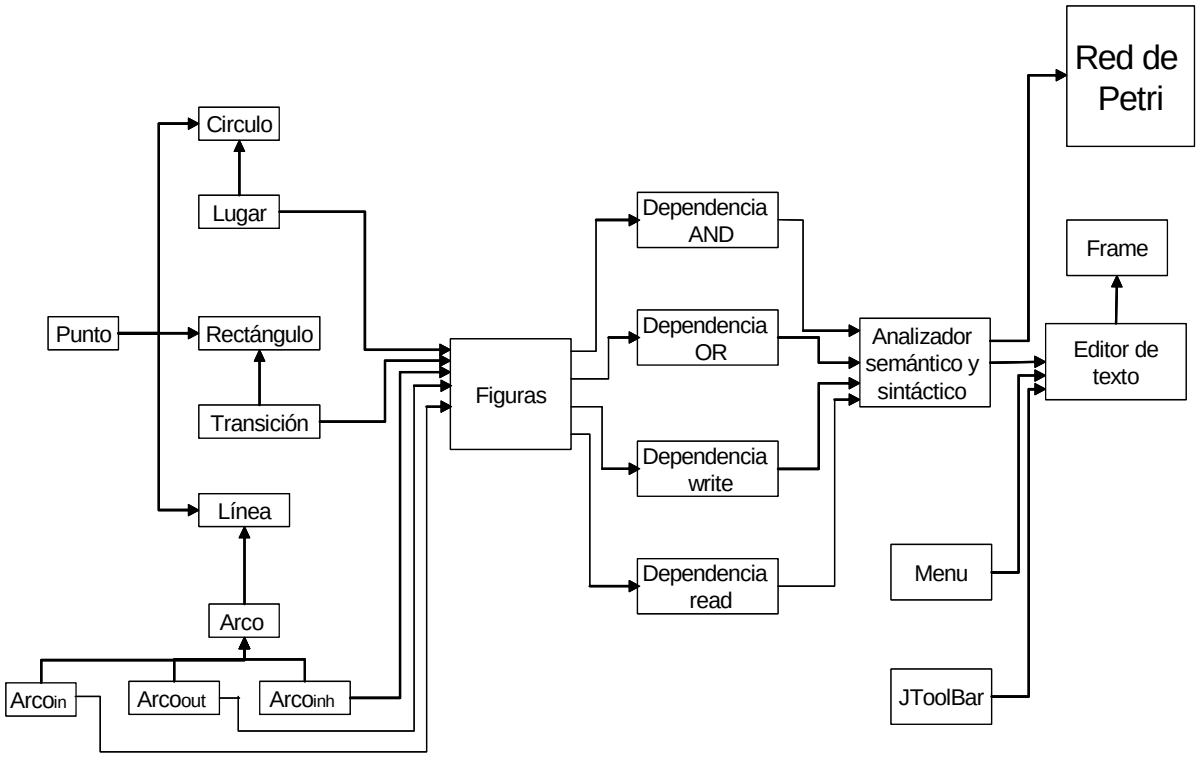

Figura 5.4: Diagrama de clases

La clase estética *Analizar*

Esta clase analiza el texto introducido por el usuario. En primer lugar almacena la información que el usuario introdujo en una variable de tipo texto. Esta es tratada por la clase de Java StringTokenizer (para utilizar esta clase se debe importar de java.util). La cual descompone la cadena extrayendo las palabras que lo conforman. Lo primero que realizamos es crear un objeto de esta clase con el constructor:

StringTokenizer st = new StringTokenizer(s);

donde s es el texto que dividimos en tokens (o cadenas) y st es el objeto de la clase StringTokenizer.

Esta clase implementa la *interface Enumeration,* por lo tanto define los métodos *nextElement* y *hasMoreElements.*

```
public class String Tokenizer implements Enumeration{
    /*  ....  */
public boolean hasMoreElements() {
    /*  Método que retorna false cuando no existen mas tokens */
  }
 public Object nextElement() {
    /*  Método para extraer tokens  */
  }
}
```

Con esta clase y sus métodos se hace el análisis gramatical y semántico. Las base para este estudio es la sintaxis del capitulo 3. Donde las reglas principales son las siguientes:

1. El nombre de las transacciones inicia con la letra mayúscula T, seguida de un número u otra letra (mayúscula o minúscula).
2. Los términos AND y OR deben ir con mayúsculas.
3. La operación *write* y *read* deben ser minúsculas.

Para la construcción de una oración correcta se deben seguir las siguientes observaciones:

- Para las dependencias de lectura y escritura:

Se coloca la operación (write o read) seguida del nombre de la transacción y de la tabla u objeto en donde se realiza la operación.

- En la dependencia de datos:

Se coloca el nombre de una transacción seguida de la dependencia commit o abort. Entre paréntesis se colocan dos transacciones y un termino AND o OR. Si se desea simular mas transacciones se pueden colocar entre paréntesis respetando que cada paréntesis puede contener dos transacciones y un termino. La unión de estos paréntesis se uno con términos AND o OR según se desee.

Las reglas antes mencionadas se deben tener en cuenta para el correcto desempeño de la aplicación. Por último si la oración es gramaticamente correcta se prosigue a generar la red de Petri. Para esto indicamos cuantas transacciones existen cuales serían sus lugares de salida y de entrada; además de los arcos que las conectan.

La clase estética Commit

Esta clase permite agregar el símbolo reservado del commit ($\longrightarrow$) al área de texto diseñada en la clase Editor.

La clase estática Abort

Esta clase permite agregar el símbolo reservado del abort ($\longleftarrow$) al área de texto diseñada en la clase Editor.

5.4.2. La clase RedPetri

En esta clase se realiza un análisis semántico para realizar la simulación con redes de Petri. Por semántica entendemos el significado de cualquier sentencia sintacticamente correcta y escrita en un determinado lenguaje. El lenguaje ya lo presentamos, por ende nos falta su interpretación. Esta se realizará por medio de una gráfica. Es decir, si una sentencia no se encuentra bien formulada la gráfica resultante será incorrecta y por ende su análisis.

El primer paso que se realiza es identificar la dependencia que existe entre las transacciones. Por ejemplo para la siguiente sentencia:

$T_i \longrightarrow (T_j \ AND \ T_x)$

T_j y T_x se deben ejecutar primero que la transacción T_i. Dentro del estudio de la sentencia detectamos que tipo de dependencia existe. Para este caso existe una dependencia AND entre las transacciones T_j y T_x. Por lo tanto creamos un objeto de tipo AND. Recordemos que existe además la dependencia OR, *write* y *read*.

Por último es importante mencionar que en esta clase se crea un área de dibujo en la cual se plasmará la red de Petri.

La clase And

Esta clase tiene como objetivo crear objetos de diferente tipo de tal suerte que pueda generar gráficamente la figura que se muestra en la Figura 4.3. Para esta clase se crean tres objetos de tipo transición. Donde dos son las que componen el bloque AND y una que representa la conjunción AND. Se crean varios objetos de tipo lugar que corresponderán a los lugares de entrada y salida de las tres transiciones en la Figura 4.3 los lugares de salida de las transacciones serán los lugares de entrada de la transacción AND (por proporcionar un ejemplo). Por último se crean objetos de tipo arco_{in} y arco_{out} para unir las transiciones y los lugares.

public draw() Este método tiene como argumentos las coordenadas donde dibuja los objetos que se crean en la clase And.

La clase Or.

Esta clase genera gráficamente la Figura 4.4, para esto se crean dos transiciones que representan las transacciones que componen el bloque. Se crean dos lugares de entrada para cada transición y un lugar especial el cual representa la dependencia OR. Además se crean los objetos de tipo arco (de entrada y salida) para conectar los lugares con las transiciones.

public draw() Este método tiene como argumentos las coordenadas donde dibuja los objetos que se crean en la clase Or.

La clase Write

Como mencionamos en el capitulo 2 la escritura de un dato (o una tabla) requiere hacer una petición por su candado. Si el candado esta disponible podemos realizar la operación de escritura. En caso contrario tendrá que esperar por el candado. Para modelar esta situación se crea un objeto

de tipo *write* el cual tiene como objetivo generar una figura como la que se muestra en la Figura 4.1. Para esto se crea un objeto de la clase Lugar$_{att}$.Para que la operación de escritura se ejecute es necesario que el Lugar$_{att}$ tenga por lo menos un token.

La clase Read

La operación de lectura a diferencia del de escritura no requiere el candado del dato para que pueda ejecutar su operación. Por lo tanto esta clase se encarga de generar en primer lugar un lugar con el dato compartido (Lugar$_{att}$) y un arco inhibidor que conecte a este lugar con la transacción que desee ejecutar la operación de lectura. Ver la Figura 4.2

5.4.3. La clase Figura

Para poder generar la red de Petri se requiere crear objetos de diferente tipo. Es decir Obejtos de tipo Lugar, Arco$_{in}$, Arco$_{out}$, Transición, etc. dependiendo de la sentencia que el usuario desee modelar. La clase Figura recibe como argumentos cuantos objetos y de que tipo necesita crear para poder generar la red de Petri.

La clase Punto

Esta clase construye objetos compuestos por dos coordenadas (x,y). Los métodos de esta clase son:

public int valorX( ): Devuelve el valor de la coordenada x del objeto

public int valorY( ): Devuelve el valor de la coordenada y del objeto

public void asigX( ): Reasigna el valor de la coordenada x del objeto

public void asigY( ): Reasigna el valor de la coordenada y del objeto

La clase Circulo

En esta clase se construyen objetos de tipo punto (x,y) que representa el origen del circulo y un valor de radio.

public Punto obtOrigen( ): Devuelve la coordenada de origen (x,y).

public int obtRadio( ) :Devuelve el valor del radio

La clase Lugar

Esta clase instancea objetos que representan un lugar en las redes de Petri. Se declaran variables para almacenar el nombre del lugar y si hace la representación de un lugar compartido.

public void nombre( ): Le asigna un nombre al lugar.

public String nombre( ): Obtiene el nombre del lugar.

public void asigDato( ): Le asigna un dato para ser representado como lugar compartido.

La clase Rectangulo

Con esta clas obtenemos objetos de tipo rectangulo. Se almacena la coordenada de origen, base y altura.

public Punto obtOrigen( ): Devuelve la coordenada de origen (x,y) del rectángulo.

public Punto obtCentro( ): Devuelve la coordenada donde se encuentra el centro del rectángulo.

public int obtAl( ): Obtiene la altura del rectangulo.

public int obtBas( ): Obtiene la Base del rectangulo.

public void asigAl( ): Asigna la altura del rectangulo.

public void asigBas( ): Asigna la Base del rectangulo.

La clase Transicion

Esta clase instancia objetos que representan un transiciones en las redes de Petri. Se declaran variables para almacenar el nombre de la transición.

public void nombre( ): Le asigna un nombre a la transición.

public String nombre( ): Obtiene el nombre de la transición**.**

La clase Linea

En esta clase se crean objetos con un Punto inicial y final los cuales forman el objeto Línea.

public Punto obtP1( ): Devuelve la primer coordenada de la línea.

public Punto obtP2( ): Devuelve la segunda coordenada de la línea.

public void asigP1( ): Asigna el valor de la primera coordenada de la línea.

public void asigP2( ): Asigna el valor de la segunda coordenada de la línea.

La clase Arco, Arco$_{in}$, Arco$_{out}$ y Arco$_{inh}$

En esta clase se definen las propiedades de los arcos en las redes de Petri. Las clases Arco$_{in}$, Arco$_{out}$ y Arco$_{inh}$ heredan las propiedades y atributos de la clase Arco. Básicamente estas clases generan objetos que conectan transiciones con lugares, lugares con transiciones y los arcos inhibidores.

public void nombre( ): Le asigna el nombre al arco.

public String nombre( ): Obtiene el nombre del arco.

public int obtLugar( ): Obtiene la coordenada del lugar que conecta.

public void asiLugar( ): Asigna el valor de la coordenada que conecta.

public int obtLugar( ): Obtiene la coordenada del lugar que conecta.

public void asiLugar( ): Asigna el valor de la coordenada que conecta.

5.5. Uso de TRANSimul

Expliquemos el funcionamiento de TRANSimul con el siguiente ejemplo:

Ejemplo 5.1 *Imaginemos que deseamos modelar el siguiente conjunto de transacciones:*

write T_i x

read T_j x

$Ti \longrightarrow (T_j\ AND\ T_x)$

Para esto en el Área de Edición el usuario introducirá el grupo de transacciones que desee simular. Como se muestra en la Figura 5.5

Ya que tenemos capturada correctamente el conjunto de transacciones y queramos generar la red de Petri se debe dar click al icono que se muestra en la Figura 5.6

La pantalla que sigue es donde se muestra la red de Petri que modela el conjunto de transacciones del ejemplo anterior y la matriz de incidencia del grupo de transacciones que se modelaron. Ver la Figura 5.7

La última pantalla muestra el resulta obtenido del análisis hecho las rutas y se le indica al usuario si existe la posibilidad de un candado mortal. Esto lo podemos ver en la Figura5.8

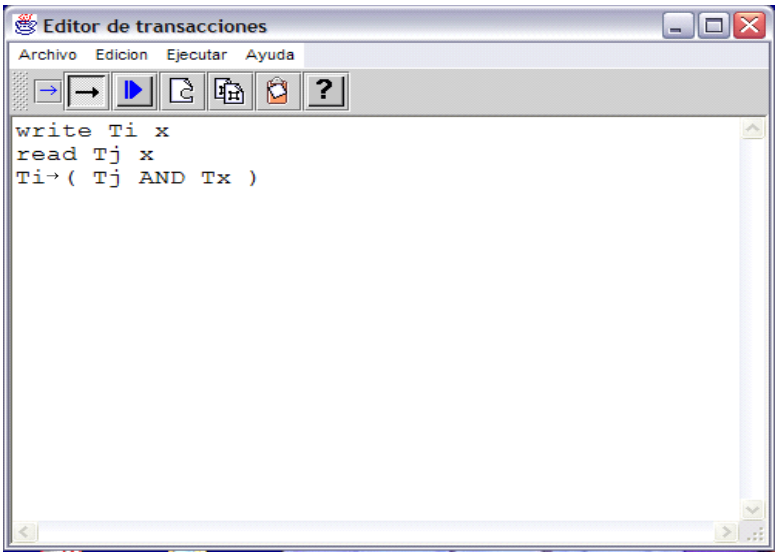

Figura 5.5: La dependencia: $T_i \longrightarrow (T_j\ AND\ T_x)$

Figura 5.6: Generando la Red de Petri

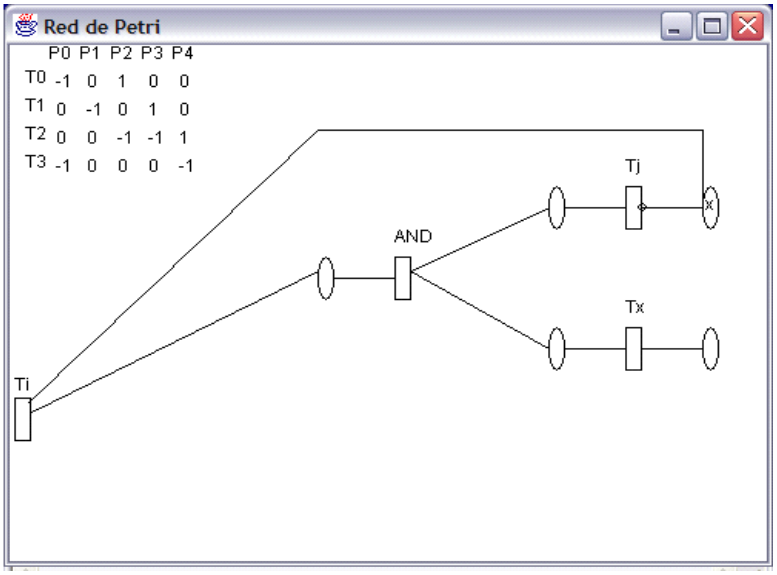

Figura 5.7: Red de Petri y matriz de incidencia generada.

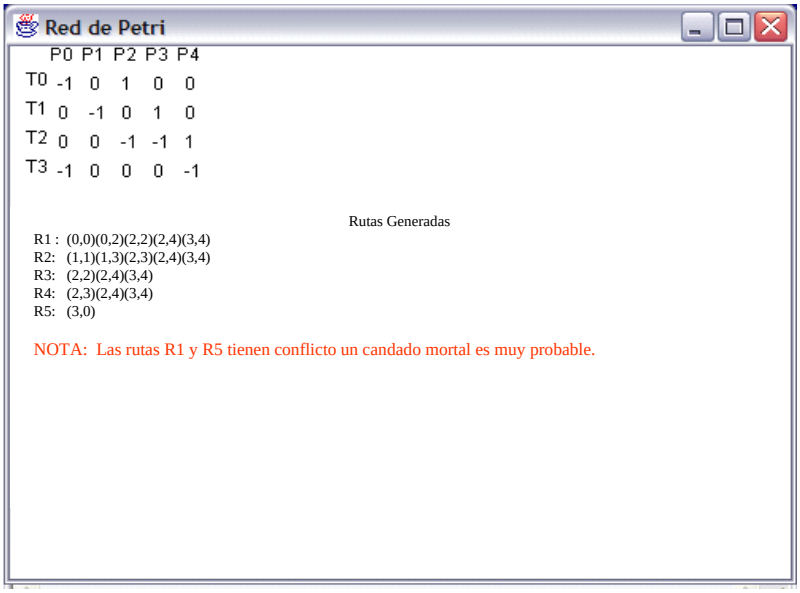

Figura 5.8: Gráfica que muestra el resultado del análisis de las rutas

Capítulo 6

Conclusiones

La mayoría de los sistemas tiene como un objetivo la eficiencia, esto es, ejecutar las instrucciones o programas en el menor tiempo posible y por supuesto exitosamente, el pensar que un sistema cumpla con éstas características resulta utópico puesto que se requiere de muchas condiciones y precondiciones para su correcto desempeño. La concurrencia y el paralelismo le han dado a la computación las técnicas necesarias para hacer mas eficiente a los sistemas pero cuenta con una gran desventaja, como sabemos los procesos o transacciones para su ejecución requieren de acceder a memoria para leer un dato o para escribir en el; ésto hace difícil la tarea porque si no controlamos correctamente los accesos podemos caer en incongruencia de los datos, por ejemplo, imaginemos que el proceso_1 necesita leer la cuenta de cheques Banco1 y el proceso_2 tiene que modificar la misma cuenta para abonarle cierta cantidad de dinero, a nadie le gustaría que al revisar tu dinero el proceso_2 no haya hecho el abono. Las redes de Petri son una herramienta fiable para modelar procesos concurrentes y paralelos, además, cuenta con técnicas muy fiables para su análisis.

Una de las técnicas que mas se utilizan para la detección de candados mortales es *WFG*. Sin embargo esta tiene los siguientes problemas:

- El análisis se realiza directamente en la gráfica (buscando ciclos en está) haciendo su estudio muy complejo. Por otro lado la gráfica tiene un crecimiento significativo complicando aun más su análisis y su manejabilidad.

- La gráfica *WFG* es estática. Esto es, no puede representar la ejecución a posteriori de un

proceso.

- La reutilización de *WFG* es muy pobre.

Una de las Características de las redes de Petri es la habilidad que tienen para simular procesos concurrentes o paralelos. Por lo tanto, la simulación de transacciones en Base de Datos resulta sencilla y práctica. Con respecto a el análisis, resulta eficaz puesto que cuenta con herramientas matemáticas que nos permiten estudiar la estructura de la gráfica y poder predecir el comportamiento de las transacciones. Con esta información podemos decidir si una transacción no tendrá problemas durante su ejecución. Por lo tanto podemos en determinado momento "podar" la gráfica y por ende su matriz de incidencia.

En esta tesis se trabajo la detección de candados mortales en sistemas centralizados. Es decir, todas las redes de Petri generadas se hacen de manera local y a posteriori de su ejecución. Básicamente evitamos la aparición de candados mortales.

Una de las grandes ventajas con respecto a [1] es el hecho de generar directamente la red de Petri a partir del conjunto de transacciones, además se considera una sintaxis mas completa que la propuesta por ellos. Por último el análisis lo realizamos en la estructura de la gráfica cuando en [1] lo realizan detectando transacciones muertas. Por lo tanto es mas completo el estudio que nosotros realizamos.

6.1. Resultados obtenidos

La detección de candados mortales con redes de Petri se utilizan ampliamente en sistemas de manufactura. Una de las investigaciones realizadas para Base de Datos es la desarrollada por [1], en está utilizan WFG para modelar un conjunto de transacciones y después mapean *WFG* a redes de Petri. En está tesis modelamos directamente con redes de Petri, proponiendo una sintaxis mas completa que la propuesta por Bertino. Ademas se desarrollo una aplicación para mostrar gráficamente la simulación.

Se propone un análisis de candados mortales utilizando la matriz de incidencia de la red de Petri obtenida. Este estudio de la estructura de la red nos arroja como resultado todas las rutas que las transacciones pueden seguir para culminar su ejecución. El algoritmo genera varias rutas, sin embargo no todas son tomadas en cuenta para el estudio. Esto es, porque nos interesan las

rutas que abarcan toda la red. Por ejemplo el análisis de la Figura 4.14 muestra 5 rutas posibles, pero no todas nos son útiles. Las que proporcionan información importante son las rutas: $R1$, $R2$ y $R5$. Donde $R3$ y $R4$ son complemento de las dos primeras rutas. En base a estas rutas se genera el estudio.

Las redes de Petri por lo tanto son una alternativa eficiente, eficaz y rápida para la detección de candados mortales.

6.2. Trabajo futuro

Los sistemas de base de datos en un futuro tendrán que soportar múltiples usuarios y, por, ende, una cantidad considerable de transacciones concurrentemente, los esquemas de control de concurrencia se hacen indispensable, pero los candados mortales necesitan mejores algoritmos, soluciones más eficientes y, sobre todo que no generen tráfico. Las redes de Petri nos brindan una técnica para mejorar los algoritmos existentes.

Como trabajo futuro pretendemos lo siguiente:

- Generar en tiempo de ejecución la detección de candados mortales, utilizando los logs de los sistemas como información para generar la red de Petri que nos permita investigar si el sistema se encuentra bloqueado.
- Poner en práctica esta investigación pero en sistemas distribuidos. Utilizar la tecnología de agentes para generar una gráfica del comportamiento general del sistema.
- Utilizar redes de Petri coloreadas para realizar una gráfica que contenga información de las actividades de cada transacción. Es decir, para cada transacción manejaremos, por ejemplo la siguiente información: número de candados adquiridos, número de conflictos, número de bloqueos, etc. Con esta información la decisión de que transacción tiene que reiniciarse será mas objetiva.

Bibliografía

[1] Bertino, E., Chiola G., Mancini L.V., Deadlock detection in the face of transaction and data dependecies. *Lecture Notes in Computer Sciencie*, vol. 1420, Springer Verlag, 1998, pp. 266-285

[2] Silberschatz A., Korth H.F., Sudarshan S., *Database System Concepts*, Third Edition, McGraw-hill, 1999

[3] Date C.J.,"*An Introduction to Database Systems",*Addison-Wesley,Sixth Edition,System Programing Series, Reading, (MA) 1995.

[4] David A., Alla H, *"Petri Nets grafcet"*, Prentice Hall,1992.

[5] Murata T., Petri Nets: Properties, analysis, and aplication, *Proceedings of the IEEE* 77(4), 1989, pp.541-580

[6] Knapp E., Deadlock Detection in Distribuited Database, *ACM Computing Surveys*, 19(4), 1987

[7] Brzezinski J., Helary J.H., Raynal M., Singhal M., Deadlock Models and a General Algorithm for Distribuited Deadlock Detection, *Journal of parallel and distributed computing,* vol. 31, 1995, pp. 112-125

[8] Ho Gary S. and Ramamoorthy C.V., Protocols for Deadlock Detection in Distributed Database Systems, *IEEE Transactions on Software Engineering*, 8(6), 1982

[9] www.daimi.au.dk/PetriNets/tools/quick.html

[10] Bhargava B., Concurrency Control in Database Systems, *IEEE Transactions on Knowledge and Data Engineering*, 11(1), 1999

[11] Medina J.: Red de Petri Coloreada Condicional y su Aplicación en Sistemas de Bases de Datos Activas, *Tesis de Maestría* presentada en la sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Investigación y de Estudios Avanzados del IPN, México. Septiembre de 2002.

[12] Meneses A., "PetrA: Herramienta para la modelación y simulación de redes de Petri", *Tesis de Maestría* presentada en la sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Investigación y de Estudios Avanzados del IPN, México. Febrero de 2002.

[13] Krivocapic N., Kemper A., Gudes E., Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis, *The VLDB journal*, vol. 199, pp. 79-100.

[14] Yao W., Perrizo W., He X., An improved Algorithm for Concurrency Control in Distributed Database Systems, *Informatics and Computer Scienc*e, 1997.

[15] Chandy K. M., Lamport L, *Distributed snapshots: Determining global state of distributed systems.* ACM Trans. Comput. Systems 3. 1985- 63-75.

[16] Mitchell D., Merritt M. J., *A distributed algorithm for deadlock detection and resolution.* 3rd ACM Symposium on Principles of Distributed Computing. New York. 1984. 282-284.

[17] Kumar V., KRISHNA- an efficient concurrency control algorithm based on dynamics attributes of transactions and its performance, *Data & Knowledge engineering*, 1997, pp. 281-296

[18] Triantafillou P., An approach to deadlock detection in multidatabase, Journal22(1), 1997, pp 39-55

[19] Jensen K.:A brief introduction to Coloured Petri Nets,Journal Computer Science Department, University of Aarhus, Denmark.

[20] Jensen K.: Coloured Petri Nets, Computer Science Department, University of Aarhus, Denmark. http://www.daiami.auu.dk/■kjensen.

[21] Jensen K., "An Introduction to the Theoretical Aspects of Colored Petri Nets". *Lecture Notes in Computer Science: A Decade of Concurrency*, vol. 803, edited by J. W. de Bakker, W.-P. de Roever, G. Rozenberg , Springer-Verlag, 1994, pp. 230-272

[22] http://www.mysql.com/

[23] http://www.oracle.com/

[24] http://www.borland.com/interbase/

[25] http://www.postgresql.org/

[26] http://www.daiami.au.dk/PetriNets/tools/

[27] Froufe A.,*JAVA 2 Manual de usuario y tutorial 2ªEdición*, Alfaomega Ra-Ma, Septiembre 2000