Facultad de Ingeniería
Escuela de Ingeniería Informática

CERTAMEN 1 LENGUAJES DE PROGRAMACIÓN: PETRINETS

Por

Eugenio Cortés
Mauricio De Juan
Miguel Yáñez

Profesor: Alonso Inostrosa Psijas
Noviembre 2021

Resumen

Las redes de Petri (PN, Petri Nets) fueron definidas por Carl Petri en 1962 [1], y corresponden a un tipo particular de grafo dirigido que, gracias a su representación matemática y gráfica, permite modelar sistemas concurrentes, paralelos y/o distribuidos [2]. Una red de petri se compone con nodos, transiciones, arcos y tokens.

El potencial de las redes de petri para representar sistemas discretos complejos es muy grande, por lo que para demostrar un buen trabajo hemos realizado dos operaciones "Fork" y "Exclusión Mutua", que serán explicadas más adelante.

También se implementó un lenguaje de programación que permite la creación y manipulación de PN. Contiene instrucciones básicas que dejan a la creación de elementos básicos compuestas anteriormente mencionados de manera independiente.

Considerando que tanto los places como las transiciones poseen nombre y tanto las transiciones como los arcos pueden requerir más de 1 token para poder ser disparados, lo que debe ser un parámetro de la instrucción correspondiente, permitirá que lo solicitado sea completado exitosamente.

Índice general

Capítulo 1

Definición del Problema

1.1. Formulación del Problema

Según lo explicado en el resumen, el profesor Alonso Inostrosa ha solicitado realizar un lenguaje de programación que permita la creación y manipulación de redes de Petri. Debe contener instrucciones básicas que permiten la creación de elementos básicos (places, arcos, transiciones y tokens) de manera independiente. Hay que considerar que tanto los places como las transiciones poseen nombre. Tanto las transiciones como los arcos pueden requerir más de un token para poder ser disparados, lo que debe ser un parámetro de la instrucción siguiente.

Pero, además, se deben crear dos instrucciones que permitan crear algún tipo de redes complejas conocidas (las que serán mencionadas en la solución propuesta) que utilicen algún tipo de parámetros. Hay que considerar también una instrucción que permita saber si, dados dos places cualquiera, existe un camino (independiente de la catidad de tokens requeridos) que los una topológicamente.

Por último, hay que realizar una red de ejemplo que también, estará en este informe.

Capítulo 2

Diseño de la solución

2.1. Solución del problema

La propuesta de la solución vinculada a la problemática planteada anteriormente, describe en utilizar un lenguaje de programación, el cual se usaron las herramientas Flex y Bison (Yacc), para crear las redes de Petri.

En el programa que se creó, contiene formulaciones utilizadas para este trabajo que representan el lenguaje que se debe escribir en los casos que se van a utilizar en los ejemplos de ejecución en la sección de experimentación (también esto será detallado en la parte de la implementación), el cual se divide en tres partes:

- "petrinet.l" (Parte Flex)

  1. Podemos crear Nodos o Places con "P" o "PLACES", retorna la regla "place".
  2. Podemos crear transiciones con "T" o "TRANSITION", retorna la regla "transition".
  3. Podemos crear arcos con "A" o "ARC", retorna la regla "arc".
  4. Podemos disparar una red con "FIRE", retorna la regla "fire".
  5. Podemos mostrar las cosas creadas con "SHOW", retorna la regla "SHOW".
  6. Podemos crear un ejemplo de exclusión mutua con "EX_MUTUA", retorna la regla "EX_MUTUA".
  7. Podemos crear un ejemplo de fork con "FORK", retorna la regla "FORK".
  8. Con un espacio, hecho de esta forma " " podemos separar las cosas para introducirlas en el otro programa descrito, retorna la regla "spc".

9. Con la palabra "FROM" podemos buscar el nodo que necesitamos chequear para buscar un camino correspondiente (si es que existe), retorna la regla "FROM".
10. Con la palabra "TO" podemos buscar el nodo con el que usamos la palabra "FROM" si es que existe un camino, esto será descrita en su sintaxis en Bison (Yacc), retorna la regla "TO".
11. Otras cosas se realizan con Bison (Yacc), ya sean para los nombres que se retornan como "name", y como "NUMERO".

- "petrinet.y" (Parte Bison o Yacc), en este apartado solo se señalarán las reglas de producción utilizadas, y luego en la sección de formulaciones utilizadas serán descritas las funciones que describe nuestro trabajo:

  1. Para crear un lugar se pone "P"o "PLACE" "nombre" "N° de tokens", ejemplo de esto "PLACE p1 2".
  2. Para crear una transición se pone "T" o "TRANSITION" "nombre", ejemplo de esto "TRANSITION t1".
  3. Para crear un arco se pone "A" o "ARC" "nombre" "nombre" donde el primer nombre corresponde al nombre del place o de la transición y el segundo al nombre del place o de la transición, contrario a la opción que se colocó al inicio. Ejemplo de esto: "ARC p1 t1".
  4. Para disparar una transición se escribe "FIRE"
  5. Para mostrar la red se escribe "SHOW"
  6. Para mostrar un primer modelo prefabricado se antepone un EX_MUTUA, un espacio " " y si queremos colocamos el número, ejemplo de esto "EX_MUTUA 4"
  7. Lo mismo sucede con "FORK" y si queremos colocamos el número, ejemplo "FORK 3"
  8. Con la regla: "FROM spc name spc TO spc name" podemos buscar si existe un camino independiente de la cantidad de tokens mostrados" por ejemplo "FROM p1 TO t1" (spc es un espacio dentro de la regla mencionada).

- "Makefile": esto solo contiene los archivos para compilar el codigo mostrado a continuación:

```
petrinet: petrinet.l petrinet.y
        bison -d petrinet.y
        flex petrinet.l lex.yy.c
        gcc -o $@ petrinet.tab.c lex.yy.c
clean:
        rm petrinet.tab.* petrinet lex.yy.c
```

2.2. Formulaciones (Código) Utilizado

Lo que vamos a describir a continuación son todas las funciones, estructuras y lenguajes utilizados en Bison o Yacc que corresponden a código C :

- Con esto definimos el objeto para los places o lugares:

```
typedef struct{
  char name[100];
  int value;
  activated_already;
}PLACE;
```

- Con esto definimos el objeto para las transiciones:

```
typedef struct{
   char name[100];
   int canfire;
   }TRANSITION;
```

- Con esto definimos el objeto para los arcos que van desde un lugar a una transicion:

```
typedef struct{
   int value;
   PLACE *from;
   TRANSITION *to;
   int buscado;
   }PTARC;
```

- Con esto definimos el objeto para los arcos que van desde una transicion a un lugar:

```
typedef struct{
    int value;
    TRANSITION *from;
    PLACE *to;
    int buscado;
    }TPARC;
```

- Con esto llevamos el contero de las entidades por cada tipo para no tener que medir el tamaño de los arreglos por cada FOR:

```
int nPlaces = 0;
int nTransitions = 0;
int nPtarcs = 0;
int nTparcs = 0;
```

- Con esto definimos los arreglos en donde se almacenarán los objetos:

```
PLACE places[100];
TRANSITION transitions[100];
PTARC ptarcs[100];
TPARC tparcs[100];
```

- Esta es la función para crear un PLACE (lugar), value es el numero de tokens y activated_already se utiliza para saber si una transición ya le quita un token para que las demas no lo hagan:

```
void addPlace(char* n, int v){
    strcpy(places[nPlaces].name, n);
    places[nPlaces].value = v;
    places[nPlaces].activated_already = 0;
    nPlaces++;
}
```

- Esta es la función para crear una TRANSITION (transición), canfire determina si la transición se puede disparar, esto para no disparla sin evaluar las demás transiciones:

```
void addTransition(char* n){
    strcpy(transitions[nTransitions].name, n);
    transitions[nTransitions].canfire = 0;
    nTransitions++;
    }
```

- Esta función determina si el usuario quiere crear un arco desde un lugar a una transición (TPA) o un arco desde una transición a un lugar (TPA) y asi asignarlo a su debido objeto. Tambien verifica que las entidades existan y que no haga un arco desde entidades del mismo tipo:

```
int checkARC(char* from, char* to){
    int fromPlace = 0;
    int toPlace = 0;
    int fromTransition = 0;
    int toTransition = 0;
    for(int i = 0; i < nPlaces; i++){
        if (!strcmp(places[i].name, from)){
            fromPlace++;
    }
        if (!strcmp(places[i].name, to)){
        toPlace++;
    }}
    for(int i = 0; i < nTransitions; i++){
        if (!strcmp(transitions[i].name, from)){
            fromTransition++;
        }
        if (!strcmp(transitions[i].name, to)){
            toTransition++;
        }
    }
    if (fromPlace == 1 && toTransition == 1){
    return 0;
    }
    if (fromTransition == 1 && toPlace == 1){
    return 1;
    }
    return -1;
    }
```

- Esta es la función para crear los arcos (ARCS), se le asignará al objeto dependiendo del resultado de la función anterior (La función aparecerá con un espacio pero quiere decir que continua:

```
void addArc(char* from, char* to, int v){
    switch (checkARC(from, to)){
    case 0:
        for(int i = 0; i < nPlaces; i++){
        if (!strcmp(places[i].name, from)) ptarcs[nPtarcs].from =
        &places[i];
        }
        for(int i = 0; i < nTransitions; i++){
           if (!strcmp(transitions[i].name, to)) ptarcs[nPtarcs].to =
           &transitions[i];
            }
        ptarcs[nPtarcs].value = v;
        ptarcs[nPtarcs].buscado = 0;
        nPtarcs++;
        break;
    case 1:
        for(int i = 0; i < nTransitions; i++){
        if (!strcmp(transitions[i].name, from))

                tparcs[nTparcs].from = &transitions[i];
             }
             for(int i = 0; i < nPlaces; i++){
                  if (!strcmp(places[i].name, to)) tparcs[nTparcs].to =
                  &places[i];
             }
            tparcs[nTparcs].value = v;
            ptarcs[nTparcs].buscado = 0;
            nTparcs++;
            break;
        case -1:
         break;
        default:
         break;
          }
        }
```

- Esta función verifica que las transiciones se puedan disparar:

```
int canFire(TRANSITION t){
     int canfire = 1;
    for (int j = 0; j < nPtarcs; j++){
     if(!strcmp(ptarcs[j].to->name, t.name)){
          if(ptarcs[j].value != ptarcs[j].from->value){
           canfire = 0;
          }
     }
    }
    return canfire;
    }
```

- Esta función dispara una transición en expecifico, quita el token, que la activó y lo mueve al lugar de destino, solo si otra transición no lo ha hecho antes, en ese caso el token se duplica como en el caso de Exclusión mutua:

```
    void fireTransition(TRANSITION t){
    //Se quita un token de los places que activaron la
transition
        for (int i = 0; i < nPtarcs; i++){
        if(!strcmp(ptarcs[i].to->name, t.name)){
            if (ptarcs[i].from->activated_already == 0){
                ptarcs[i].from->value--;
                ptarcs[i].from->activated_already = 1;
            }
            }
        }
  //Se agrega un token a los places a los que apunta la transition
        for (int i = 0; i < nTparcs; i++){
            if(!strcmp(tparcs[i].from->name, t.name)){
            if (tparcs[i].to->activated_already == 0){
                tparcs[i].to->value++;
                tparcs[i].to->activated_already = 1;
            }
        }
    }
    }
```

- Esta función principal es la que se encarga de disparar todas las transiciones que se puedan disparar, primero determina que transiciones se pueden activar y luego las dispara. Tambien se resetea la entidad activated_already de los lugares:

```
void fireNET(){
    for(int i = 0; i < nTransitions; i++){
        if (canFire(transitions[i])){
            transitions[i].canfire = 1;
        }
    }
    for(int i = 0; i < nTransitions; i++){
        if (transitions[i].canfire == 1){
            fireTransition(transitions[i]);
            transitions[i].canfire = 0;
            printf("transici n %s se activ \n",
            transitions[i].name);
        }
    }
//Para devolver todos los places a su estado normal
    for (int i = 0; i < nPlaces; i++){
        places[i].activated_already = 0;
    }
}
```

- Función para mostrar los lugares o places:

```
void showPlaces(){
 for(int i = 0; i < nPlaces; i++){
    printf("PLACE %d\n", i);
    printf("    name: %s\n", places[i].name);
    printf("    tokens: %d\n", places[i].value);
    }
 }
```

- Función para mostrar las transiciones:

```
void showTransitions(){
    for(int i = 0; i < nTransitions; i++){
        printf("TRANSITION %d\n", i);
        printf("        name: %s\n", transitions[i].name);
        }
    }
```

- Función para mostrar todas las entidades en pantalla

```
void showArcs(){
    for(int i = 0; i < nPtarcs; i++){
        printf("ARC (P->T) %d\n", i);
        printf("        from: %s\n", ptarcs[i].from->name);
        printf("        to: %s\n", ptarcs[i].to->name);
        printf("        value: %d\n", ptarcs[i].value);
    }
    for(int i = 0; i < nTparcs; i++){
        printf("ARC (T->P) %d\n", i);
        printf("        from: %s\n", tparcs[i].from->name);
        printf("        to: %s\n", tparcs[i].to->name);
        printf("        value: %d\n", tparcs[i].value);
    }
}
```

- Estas cuatro funciones solo crean nombres para las entidades de los modelos prearmados las 2 primeras corresponden al caso de exclusión mutua y el segundo para el fork, ejemplo: el usuario necesita 10 PLACES, entonces se crearian los lugares con los nombres p0, p1, p2, p3, ... , p10:

```
char* t_str(int a){
    char *buf = malloc(sizeof(char) * 100);
    snprintf(buf, 100, "EM_T%d", a);
    return buf;
}
char* p_str(int a){
    char *buf = malloc(sizeof(char) * 100);
    snprintf(buf, 100, "EM_P%d", a);
    return buf;
}
char* t_str_fork(int a){
    char *buf = malloc(sizeof(char) * 100);
    snprintf(buf, 100, "FK_T%d", a);
    return buf;
}
char* p_str_fork(int a){
    char *buf = malloc(sizeof(char) * 100);
    snprintf(buf, 100, "FK_P%d", a);
    return buf;
}
```

- Función para crear el modelo de exclusión mutua con una cantidad N de caminos alternos. Lo creamos con las funciones directamente y no con el lector de strings de Bison porque por alguna extraña y aun desconocida razón esta arrojaba syntax error antes de crear el modelo lo que hacia que el programa se detuviera despues de crearlo:

```
void create_ex_mutua(int alt){
   if (alt < 1) alt = 2;
        int left_t = 1;
        int mid_p = 2;
        int right_t = 2;
        addPlace("EM_P0",1);
        addPlace("EM_P1",0);
        addTransition("EM_T0");
        addArc("EM_P1", "EM_T0", 1);
        addArc("EM_T0", "EM_P0", 1);
        for(int i = 0; i < alt; i++){
             addTransition(t_str(left_t));
             addArc(p_str(0), t_str(left_t), 1);
             addPlace(p_str(mid_p),0);
             addArc(t_str(left_t), p_str(mid_p), 1);
             addTransition(t_str(right_t));
             addArc(p_str(mid_p), t_str(right_t), 1);
             addArc(t_str(right_t), p_str(1), 1);
             left_t+=2;
             mid_p++;
             right_t+=2;
        }
   }
```

- Función para crear el modelo Fork el cual recibe como parametro lacantidad de caminos alternos, por default serán 2:

```
void create_fork(int alt){
    if (alt < 1) alt = 2;
    int begin_places = 1;
    addPlace("FK_P0",1);
    addTransition("FK_T0");
    addArc("FK_P0", "FK_T0", 1);
    for(int i = 0; i < alt; i++){
        addPlace(p_str_fork(begin_places),0);
        addArc("FK_T0", p_str_fork(begin_places), 1);
        begin_places++;
        }
    }
```

- Las siguientes funciones sirven para buscar caminos y resetear los arcos para señalar que hay o no caminos encontrados.

```
int FOUND = 0;

int tp_path(char * from_transition_name, char * to_place_name,
char * from_place_name){
                for(int i = 0; i < nTparcs; i++){
    if (!strcmp(tparcs[i].from->name, from_transition_name) &&
    tparcs[i].buscado == 0){
        tparcs[i].buscado = 1;
        if (!strcmp(tparcs[i].to->name, from_place_name)){
            return 0;
        }
        pt_path(tparcs[i].to->name, to_place_name);
        }
        }
        return 0;
    }

int pt_path(char * from_place_name, char * to_place_name){
  if (!strcmp(from_place_name, to_place_name)){
    printf("camino encontrado\n");
    FOUND = 1;
    return 1;
  }
  for(int i = 0; i < nPtarcs; i++){
    if (!strcmp(ptarcs[i].from->name, from_place_name) &&
    ptarcs[i].buscado == 0){
      ptarcs[i].buscado = 1;
      tp_path(ptarcs[i].to->name, to_place_name, from_place_name);
    }
  }
  return 0;
}
```

```
void reset_arcs(){
  for(int i = 0; i < nPtarcs; i++){ ptarcs[i].buscado = 0;}
  for(int i = 0; i < nTparcs; i++){ tparcs[i].buscado = 0;}
  if (!FOUND) { printf("camino no encontrado\n"); }
  FOUND = 0;
}
```

- Todas las funciones descritas anteriormente sirven para la creación y manipulación de las redes de Petri, en la siguiente sección mostraremos algunos ejemplos que tienen que ver con la sintaxis, incluyendo los ejemplos de exclusión mutua y fork

Capítulo 3

Experimentación

3.1. Realizar compilación de los ejemplos (entradas)

Siempre hay que tener de referencia que para realizar las compilaciones en cada uno de los casos mostrados, debemos seguir lo siguiente:
- Escribir "make" en la consola estando en el directorio de los archivos
- Escribir "./petrinet" para iniciar el archivo ejecutable creado

Para el primer ejemplo crearemos una red de Petri simple que no sea parte de los modelos pre-fabricados como se puede ver en la siguiente imagen.

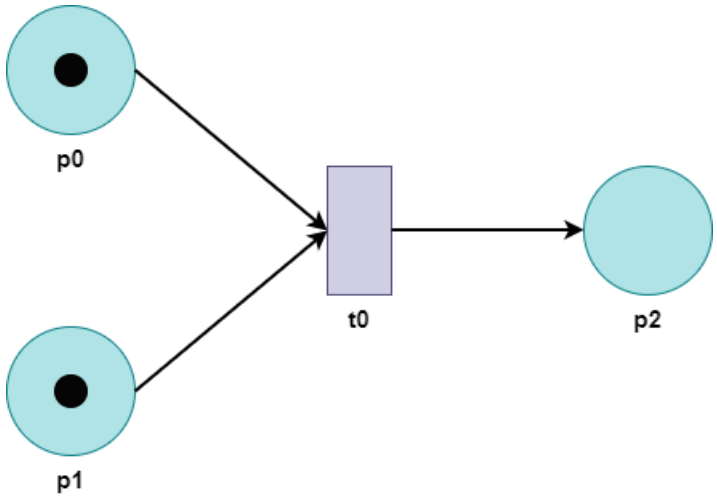

Figura 1: Red de Petri simple

Para la creación de esta red se necesitaría la siguiente sucesión de comandos:

```
P  p0  1
P  p1  1
T  t0
P  p2
A  p0  t0
A  p1  t0
A  t0  p2
```

En el segundo ejemplo se hará uso del modelo pre-fabricado de exclusión mutua y un parámetro para indicar la cantidad de caminos alternos.

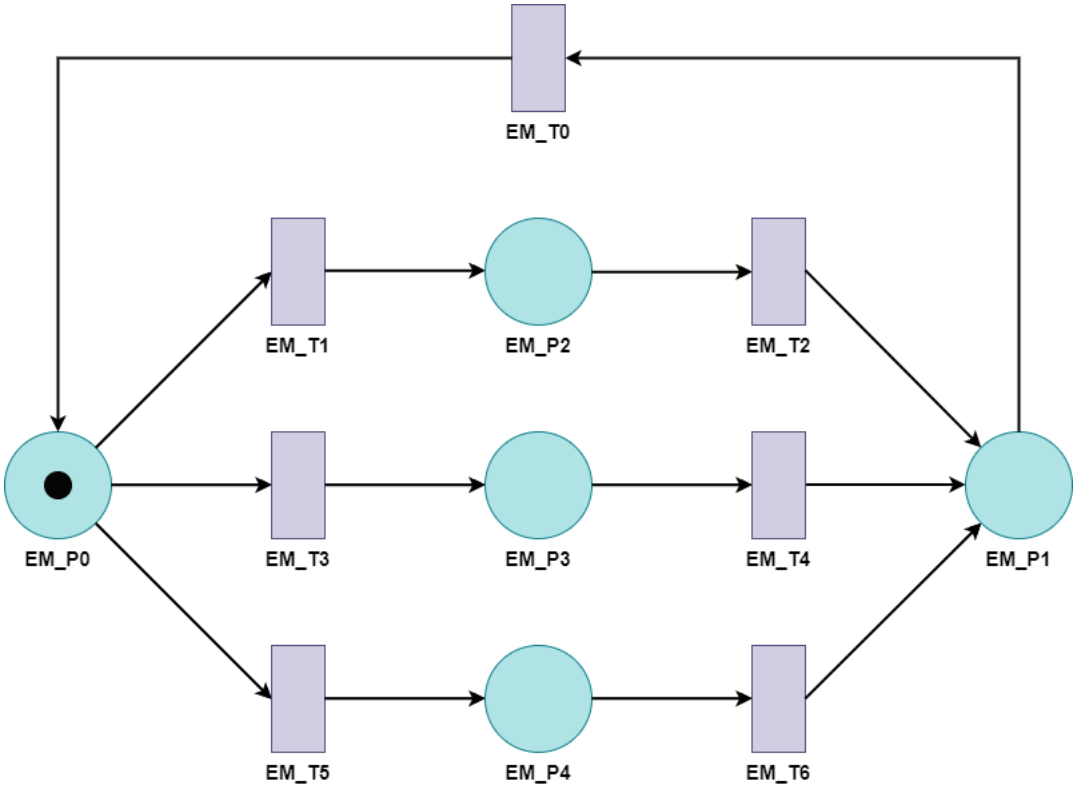

Figura 2: Red de Petri exclusión mutua

Al ser un modelo pre-fabricado, solo se necesita del siguiente comando para creaarlo.

```
EX_MUTUA  3
```

En el tercer ejemplo se hará uso del modelo pre-fabricado fork y un parámetro para indicar la cantidad de caminos alternos.

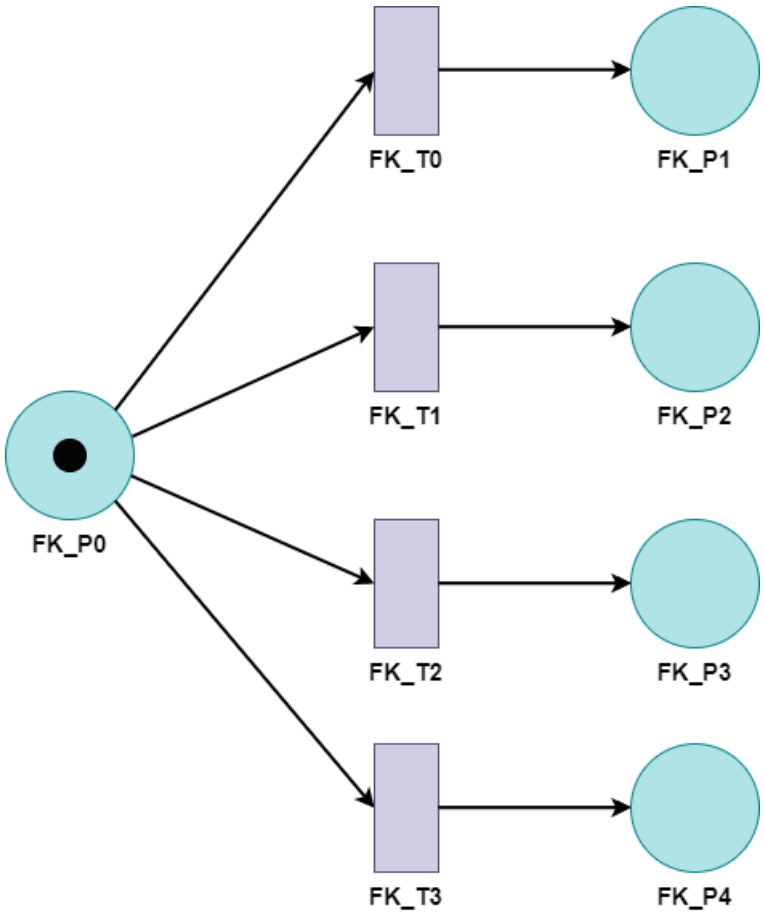

Figura 3: Red de Petri fork

Al ser un modelo pre-fabricado, solo se necesita del siguiente comando para creaarlo.

FORK 4

Finalmente, para el cuarto ejemplo crearemos una red de Petri para realizar una búsqueda de caminos entre fiderentes lugares.

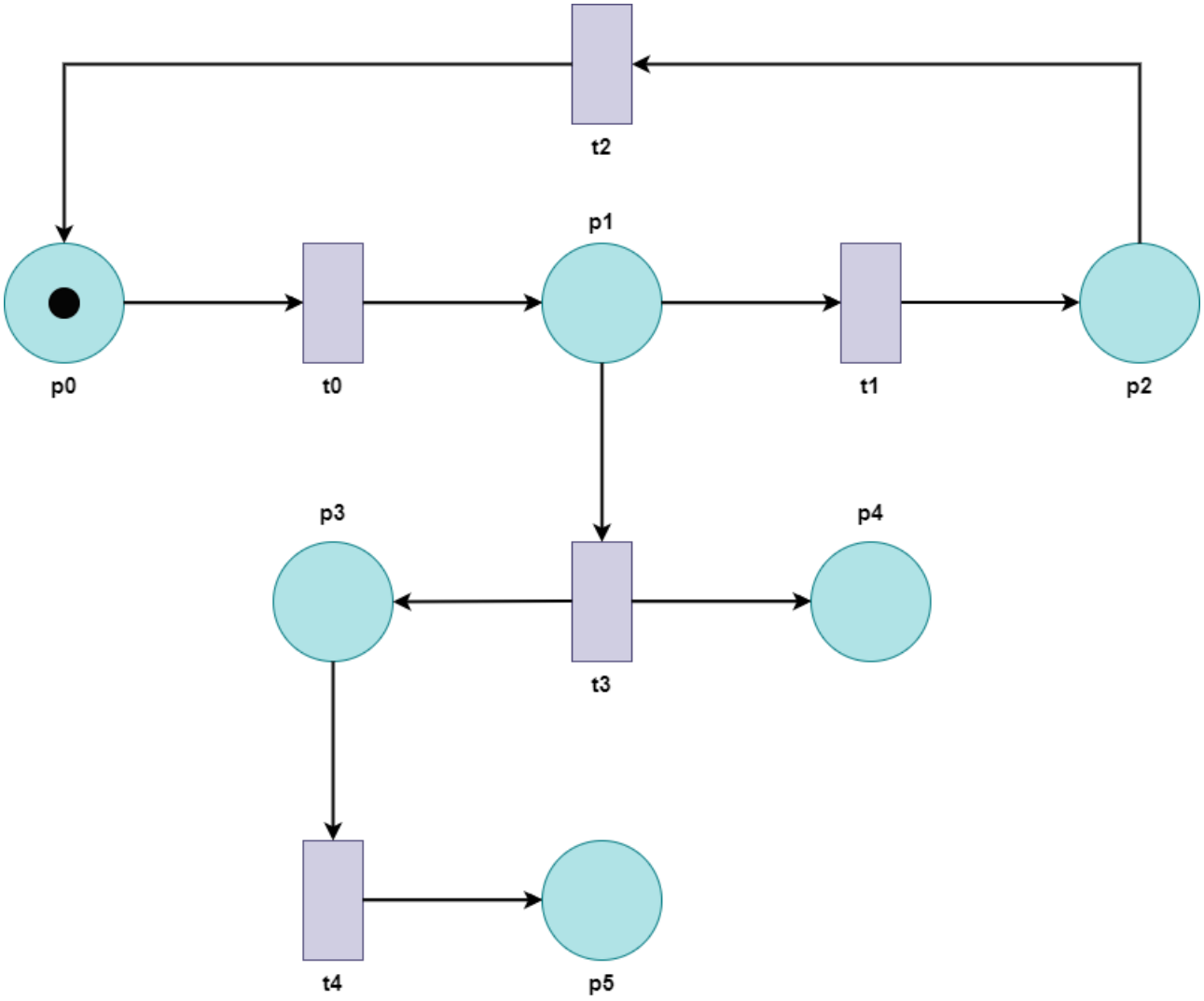

Figura 4: Red de Petri para realizar caminos

Para la creación de esta red se necesitaría la siguiente sucesión de comandos:

```
P  p0  1
P  p1
P  p2
P  p3
P  p4
P  p5
T  t0
T  t1
T  t2
T  t3
T  t4
A  p0  t0
A  t0  p1
A  p1  t1
A  t1  p2
A  p2  t2
A  t2  p0
A  p1  t3
A  t3  p4
A  t3  p3
A  p3  t4
A  t4  p5
```

3.2. Ejemplos de ejecución (salidas y esquemas para las sub redes creadas)

Una vez creado el modelo de la Figura 1, usaremos el comando "SHOW" para mostrar las entidades, como se puede ver en la Figura 5, los lugares 0 y 1 tienen un token cada uno.

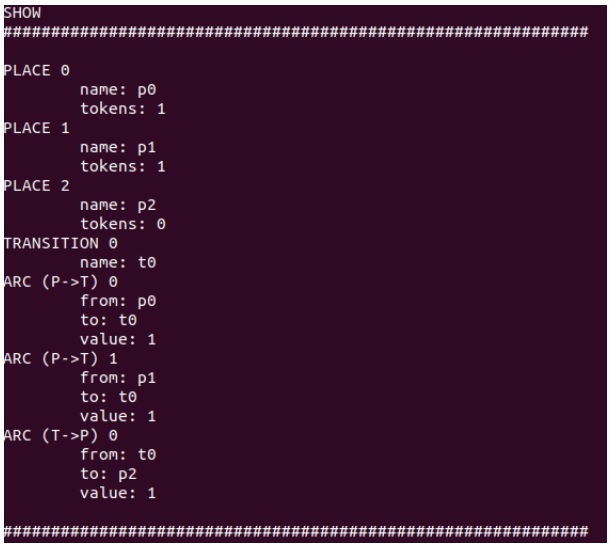

Figura 5: Entidades del ejemplo 1.

Disparamos la red utilizando el comando "FIRE" y volvemos a ver el estado de las entidades, como se puede apreciar en la Figura 6, ahora solo el lugar 2 tiene un token.

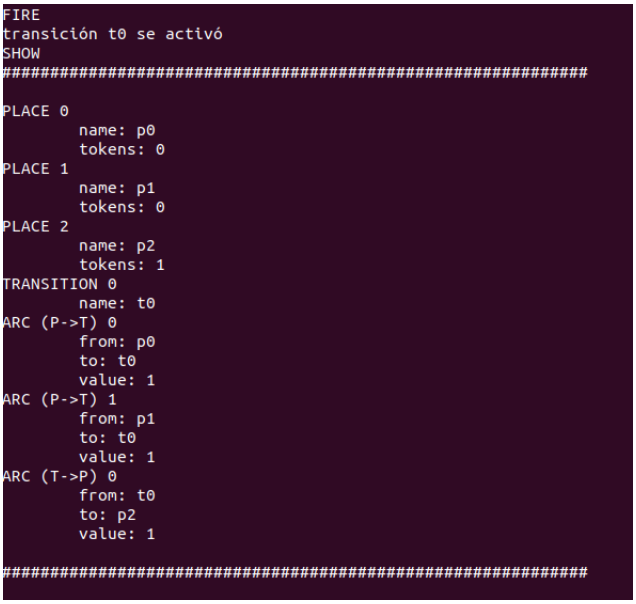

Figura 6: Entidades del ejemplo 1 despues de ser disparada.

Ahora creamos el modelo de la Figura 2, usaremos el comando "FIRE" varias veces para ver el comportamiento de la red. Como se puede ver en la Figura 7, se crea un bucle en donde primero se activan las primeras tres transiciones del los caminos alternos, luego las segundas tres y finalmente la transición que lleva el token al lugar de origen.

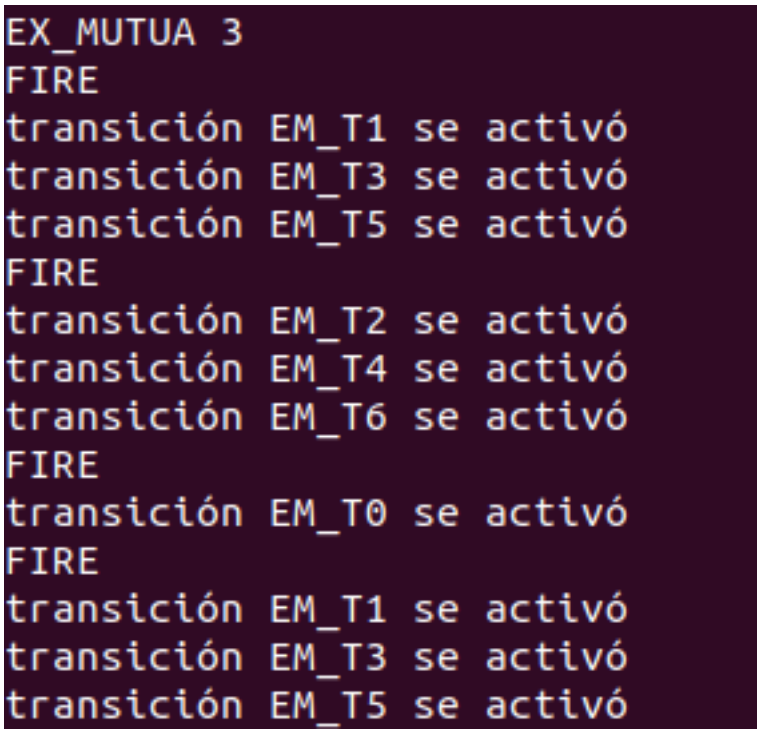

Figura 7: Comportamiento del ejemplo 2 después de ser disparada varias veces.

Seguimos con la creación el modelo de la Figura 3, usaremos el comando "FIRE" para disparar la red y comtraremos el estado de los lugares antes y después. Como se puede apreciar en la Figura 9, el token se movió del lugar de origen a todos los lugares conectados a este.

```
FORK 4
SHOW
############################################################

PLACE 0
        name: FK_P0
        tokens: 1
PLACE 1
        name: FK_P1
        tokens: 0
PLACE 2
        name: FK_P2
        tokens: 0
PLACE 3
        name: FK_P3
        tokens: 0
PLACE 4
        name: FK_P4
        tokens: 0
```

Figura 8: Lugares de un modelo Fork de cuatro caminos alternos.

```
FIRE
transición FK_T0 se activó
SHOW
############################################################

PLACE 0
       name: FK_P0
       tokens: 0
PLACE 1
       name: FK_P1
       tokens: 1
PLACE 2
       name: FK_P2
       tokens: 1
PLACE 3
       name: FK_P3
       tokens: 1
PLACE 4
       name: FK_P4
       tokens: 1
```

Figura 9: Lugares de un modelo Fork de cuatro caminos alternos después de ser disparado.

Finalmente, replicamos el modelo de la Figura 4, y buscamos si existe un camino entre el lugar "p0" y "p5".

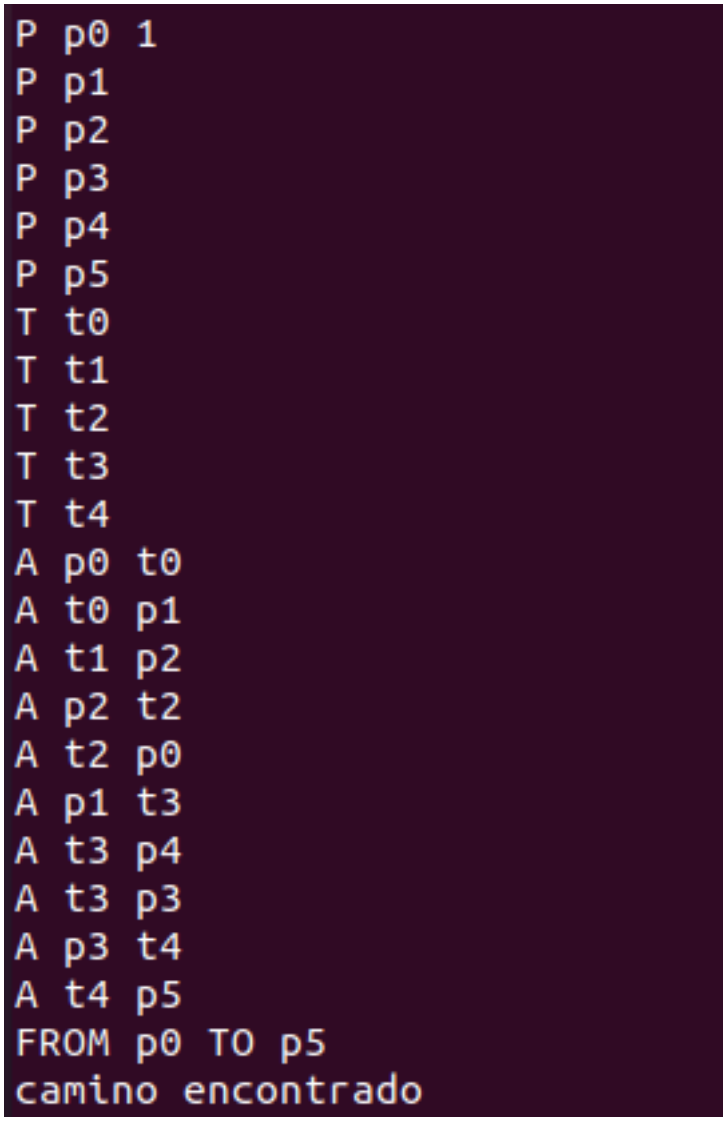

Figura 10: Búsqueda de caminos entre dos lugares.

Capítulo 4

Conclusiones

4.1. Conclusiones

Después de haber realizado la creación y manipulación de las redes de Petri se puede concluir que:

- Una de las principales dificultades con el lenguaje de Flex y Bison (Yacc), fue que al no haber suficiente documentación tuvimos que usar mucho ingenio para crear las funciones mostradas en la formulación tanto como las redes creadas como las subredes. En esa creación tuvimos que repasar muchas veces el enunciado para obtener los resultados mostrados en la experimentación.

- El lenguaje en algunos casos, tuvo fallas de forma inicial debido a que, en la creación de las subredes, obteniamos loops infinitos, pero utilizando la recursividad, logramos que no solo la exclusión mutua y fork fuesen hechos que se pueden mostrar tanto en código como en este informe, sino que también vimos resultados en la obtención de agregaciones de las redes en caminos cortos y largos según lo que necesitasemos en el momento.

- Como posible trabajo futuro, podemos pensar en la creación de redes más complejas añadiendo funciones más interesantes, si bien este certamen, por el tiempo acotado que teníamos para entregar no se logró implementar algo que fuese de mayor calibre, creemos que se puede tomar como base para la implementación de trabajos, no solo en Flex y Bison (Yacc), sino también en otros lenguajes como Java, Python, Go, entre otros.

Bibliografía

[1] C. A. Petri, “Kommunikation mit automaten [dissertation],” *Rheinisch-Westfaelisches Institut fuer Instrumentelle Mathematik an der Universitaet Bonn, Schriften des IIM*, p. 2, 1962.

[2] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.