Métodos Formales Industriales (MFI)

—prácticas—

Grado de Ingeniería en Informática

Práctica 4: Model checking en SPIN/Promela utilizando lógica LTL

Santiago Escobar

Despacho 237
Edificio DSIC 2 piso

1. Introducción

La verificación de programas consiste en determinar si una implementación o diseño satisface su especificación, es decir, es correcta respecto a los requisitos establecidos en su especificación. Esta tarea se ha desarrollado tradicionalmente sometiendo el diseño (o programa) a una batería de casos de prueba y comparando las salidas obtenidas con las esperadas. Sin embargo, conforme los sistemas han ido creciendo en tamaño, el número de posibles casos de prueba ha aumentado considerablemente, de tal forma que este estilo de verificación sólo puede cubrir una pequeña fracción de las posibles combinaciones de datos de entrada. La verificación formal de programas ("model checking") consiste en el uso de procedimientos de decisión para caracterizar un diseño por completo, es decir, equivalente a realizar una validación exhaustiva para una determinada propiedad o requisito. Este tipo de procedimientos se aplican de forma más efectiva si disponemos de herramientas automatizadas, como por ejemplo Spin.

El sistema Spin es una herramienta de validación de sistemas software distribuidos usando la lógica temporal LTL. Spin es una herramienta de validación on-the-fly, es decir, no requiere que el sistema de estados asociado al modelo sea finito, como en NuSMV, sino que el conjunto de estados alcanzables desde el estado de partida sea finito. Spin es apropiado para la verificación de software, no para la verificación de hardware. Dispone de un lenguaje de alto nivel, llamado PROMELA (a PROcess MEta LAnguage), para especificar la descripción del modelo (o sistema). Spin se ha utilizado para la detección de errores de diseño en sistemas distribuidos tales como sistemas operativos, protocolos de comunicaciones, algoritmos concurrentes, protocolos de señalizamiento en vías de trenes, etc. La NASA lo utiliza de forma rutinaria para verificar software, p.ej. en las misiones Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc.

Primero vamos a proporcionar una pequeña introducción sobre la lógica temporal utilizada por Spin para luego proceder a explicar el sistema Spin y su lenguaje Promela.

2. Linear Time Temporal Logic (LTL)

Linear Time Temporal Logic (LTL) es una lógica temporal utilizada para expresar propiedades temporales de un sistema reactivo o concurrente dentro del contexto del model checking. Utiliza proposiciones elementales, operadores lógicos y operadores temporales (de estado o de caminos).

2.1. Operadores lógicos

Los operadores lógicos son los usuales en lógica: ! (negación), | (or), & (and), $\rightarrow$ (implicación), y $\leftrightarrow$ (equivalencia). También se pueden usar los valores lógicos `true` and `false` y los operadores lógicos de comparación: = (igualdad), != (distinto), <, >, >=, <=, así como los operadores aritméticos básicos: $+$, $-$, $*$, /, *mod*.

2.2. Operadores temporales

A diferencia de CTL, toda fórmula LTL se entiende como una fórmula $A\ p$ donde p es una fórmula de camino que debe satisfacerse en todos los caminos que surjan del estado actual. Fijando un estado actual y un camino a partir de él, determinamos la satisfacibilidad de una fórmula de camino p con respecto a los estados siguientes al actual dentro del camino fijado.

- Fórmula "$X\ p$" (Next): la proposición p debe satisfacerse en el estado siguiente al actual.
- Fórmula "$G\ p$" (Globally): la proposición p debe satisfacerse en todos los estados posteriores al estado actual en el camino fijado.

- Fórmula "$F\ p$" (Eventually): la proposición p debe satisfacerse en al menos un estado posterior al estado actual en el camino fijado.
- Fórmula "$p\ U\ p'$" (Until): la proposición p debe satisfacerse en todos aquellos estados contiguos y posteriores al estado actual mientras la proposición p' no se satisfaga.

2.3. Ejemplos

Si suponemos que la proposición p describe "llueve" y que la proposición q describe "vamos a jugar al tenis", tenemos las siguientes ejemplos de fórmulas LTL:

1. $AG\ p$. Siempre llueve.
2. $AF\ p$. Siempre llegará un momento en que llueva.
3. $AFG\ p$. Siempre llegará un momento en que llueva y no pare de llover.
4. $AGF\ (!p \wedge q)$. Siempre llegará un momento en que deje de llover y juegue al tenis y esto ocurrirá un número infinito de veces.
5. $AX\ p$. Va a llover en el siguiente instante de tiempo, pase lo que pase.
6. $A\ (q\ U\ p)$. Siempre juego al tenis hasta que empieza a llover.

Hay que recordar que las lógicas CTL y LTL son incomparables. Por ejemplo,

1. $AFG\ p$. Siempre llegará un momento en que llueva y no pare de llover. Esto se puede describir con LTL pero no con CTL.
2. $AG(EF\ p)$. Para todo instante de tiempo, existe la posibilidad de que llueva. Esto se puede describir con CTL pero no con LTL.

La lógica CTL* es capaz de expresar tanto fórmulas CTL como LTL, aunque no existen técnicas eficientes de verificación.

3. El sistema Spin

El sistema Spin comprueba que un sistema software distribuido satisface la especificación asociada proporcionada en la lógica LTL. El lenguaje incluido en Spin, llamado PROMELA (a PROcess MEta LAnguage), permite describir la relación de transición dentro del sistema usando una notación imperativa parecida al lenguaje C.

Spin es una herramienta para analizar la consistencia lógica de sistemas concurrentes, específicamente de protocolos de comunicaciones o sistemas concurrentes. El sistema (o modelo) se describe en Promela. El lenguaje permite la creación dinámica de procesos concurrentes. La comunicación entre procesos se puede realizar a través de variables globales (compartidas) o a través de canales de comunicación. Los canales de comunicación se pueden definir de forma síncrona ("rendezvous"), o asíncrona ("buffered").

A partir de un modelo especificado en Promela, Spin puede realizar simulaciones aleatorias de la ejecución del sistema o puede generar un programa del lenguaje C que lleve a cabo una verificación de las propiedades del sistema (model checking). Durante la simulación y verificación, Spin comprueba la ausencia de puntos muertos (deadlocks), recepciones de datos no permitidas y código inalcanzable. El verificador puede además comprobar la corrección de invariantes del sistema,

puede encontrar ciclos inútiles de ejecución y puede verificar la corrección de propiedades expresadas con fórmulas LTL. Los invariantes se especifican con la palabra reservada `assert` mientras que las fórmulas LTL se traducen en expresiones *never*, que indican un conjunto de caminos (posiblemente infinitos) que no deben darse en el modelo. Sin embargo, en versiones modernas de Spin, se dispone de una palabra reservada `ltl` para incluir propiedades en LTL que son automáticamente traducidas a expresiones `assert` o *never*.

El verificador está configurado para ser muy eficiente y usar la mínima cantidad de memoria. Spin realiza una verificación exhaustiva que permite establecer que el sistema está libre de errores con exactitud matemática.

Spin es una herramienta muy popular, utilizada por miles de personas en todo el mundo. Fue desarrollada inicialmente por los laboratorios Bell en los principios de los años 80. La herramienta está públicamente disponible desde 1991 y continúa evolucionando para incluir los últimos avances en el área. En Abril de 2002 recibió el prestigioso premio "System Software Award" de ACM.

La herramienta está accesible en la URL:

`http://spinroot.com`

El sistema Spin está disponible para Windows y diferentes versiones de UNIX (incluyendo Linux y Mac OS X). El programa necesita un compilador de C instalado en la máquina para poder funcionar, ya que Spin genera programas C que son versiones especializadas de un model checker genérico. Su forma de uso es:

```
$ spin -a archivo
$ gcc -o pan pan.c
$ ./pan
```

donde, de esta forma, Spin primero genera un fichero `pan.c` a partir del código de Promela almacenado en el fichero `archivo`, luego compilamos el programa C generado y, por último, ejecutamos el programa `pan`, que es el model checker especializado para este modelo de Promela. En la ejecución del programa "`./pan`"podemos añadir dos opciones: "`./pan -a`"para comprobar fórmulas LTL codificadas como expresiones never y "`./pan -f`"para asegurarnos de que la verificación incluye restricciones de fairness sobre los distintos procesos.

El programa está instalado en los laboratorios docentes del DSIC en sus versiones linux y Windows. Además, existe un entorno gráfico, llamado `xspin`.

3.1. Sintaxis de Promela

Un programa en Promela consiste en procesos, canales de mensajes y variables globales (aunque los procesos pueden tener variables locales también, inaccesibles fuera del proceso en cuestión). Los procesos se entienden como objetos globales. Los canales de mensajes son variables, declaradas de forma global o local. Los procesos especifican el comportamiento, canales y variables globales que describen el entorno en el cual los procesos se ejecutarán.

En Promela, cada instrucción implica un bloqueo del proceso que la ejecuta si dicha instrucción no puede ser ejecutada inmediatamente. Por lo tanto, hay que tener cuidado al escribir instrucciones. Por ejemplo, el siguiente código se quedará bloqueado en la condición del bucle mientras `a` sea igual a `b`, es decir, la condición implica bloqueo hasta que no se satisfaga:

```
while (a != b)
   skip /* wait for a==b */
```

el mismo comportamiento se puede obtener simplemente con la expresión:

```
(a == b)
```

ya que se quedará bloqueada hasta que ambas variables tengan el mismo valor.

Las variables se utilizan para almacenar información y pueden ser globales o locales, en función de si son declaradas fuera de todo proceso o dentro de algún proceso. Las siguientes declaraciones:

```
bool flag;
int state;
byte msg;
```

definen variables que pueden almacenar valores enteros de tres rangos distintos. La siguiente tabla muestra los tipos de datos básicos disponibles en Promela:

Tipo	Rango
bit	$0, 1$
bool	*false*, *true*
byte	$0..255$
chan	$1..255$
mtype	$1..255$
pid	$0..255$
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..2^{n}-1$

Las palabras `bit` y `bool` son sinónimos para un simple bit. Una variable de tipo `mtype` define valores simbólicos. El siguiente ejemplo define una variable con cuatro posibles valores `idle`, `entering`, `critical` y `exiting`:

```
mtype = {idle,entering,critical,exiting}
```

Se pueden definir vectores de un tipo base. Por ejemplo, un vector de 2 bytes:

```
byte state[2]
```

como en el lenguaje C, los arrays empiezan por el valor 0, es decir, en el ejemplo de antes tenemos las variables *state*[0] y *state*[1], mientras que *state*[2] no existe y genera un error dinámico de ejecución.

Los procesos se definen con la palabra `proctype`. El siguiente ejemplo declara un proceso con una única variable y una asignación a dicha variable:

```
proctype A() {
byte state;
    state = 3
}
```

Se pueden describir expresiones condicionales con el símbolo "`->`". En realidad, el símbolo "`->`" representa lo mismo que el símbolo "`;`" debido a los bloqueos asociados a toda expresión. El siguiente ejemplo representa una variable global inicializada a 2 y dos procesos A y B, donde A se queda bloqueado esperando a que la variable *state* valga 1 para luego asignarle el valor 3, mientras que B no entra en bloqueo y puede ir decrementando la variable:

```
byte state = 2;

proctype A() {
  (state == 1) -> state = 3
}

proctype B() {
  state = state - 1
}
```

La palabra reservada `proctype` sólo declara el comportamiento de los procesos, no define instancias de ellos, y por lo tanto deberemos crear tantas instancias como queramos con la palabra reservada `run`. Nótese que los procesos pueden tener variables de entrada en su definición que deben ser instanciadas en el momento de llamar a `run`. Hay siempre un proceso especial llamado `init` que será el proceso de inicio del modelo (o programa).

```
proctype A(byte state; short foo) {
   (state == 1) -> state = foo
}

init {
   run A(1, 3)
}
```

Como ocurre en C, los vectores y los procesos no pueden ser pasados en las llamadas a la función `run`. En Promela, deberemos utilizar canales de comunicación para enviar datos compuestos o complejos entre procesos. Los canales representan datos compartidos de forma distribuida, síncrona o asíncrona. En esta boletín no vamos a describir los canales de comunicación ya que no son necesarios para hacer los ejercicios y la comunicación entre procesos se puede simular con variables globales.

En el caso de que varios procesos intenten acceder (o escribir) en una misma variable global, Spin generará todas las posibles combinaciones con instrucciones intercaladas. Por ejemplo, el siguiente programa puede terminar con los valores 0, 1 y 2 en la variable *state* dependiendo del orden exacto de ejecución de los dos procesos A y B:

```
byte state = 1;

proctype A() {
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}

proctype B() {
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}

init {
    run A(); run B()
}
```

En Promela evitaremos que un grupo de instrucciones sea interrumpido por la ejecución de otro proceso con la palabra `atomic`. Por ejemplo, el siguiente programa termina sólo con los valores 0

y 2, ya que ambos procesos incrementarán la variable *state* pero ninguno de ellos es interrumpido desde la condición `state==1` hasta la asignación.

```
byte state = 1;

proctype A() {
   atomic {
       (state==1) -> state = state+1
   }
}

proctype B() {
   atomic {
       (state==1) -> state = state-1
   }
}

init {
   run A(); run B()
}
```

La instrucción de selección más simple es el `if`. Por ejemplo:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

Cada selección va encabezada por `::` y la expresión `if` seleccionará aquella selecciones que no estén bloquedas. En el ejemplo anterior las dos condiciones son excluyentes y el `if` escogerá sólo una cada vez, pero no tienen porqué ser excluyentes y Spin considerará todas las posibilidades en paralelo. Si todas las guardas están bloqueadas, el `if` se bloquea a si mismo. En el siguiente ejemplo, el proceso `counter` incrementará o decrementará de forma indeterminista y Spin considerará ambas posibilidades en paralelo:

```
byte count;

proctype counter() {
    if
    :: count = count + 1
    :: count = count - 1
    fi
}
```

Una extensión del `if` es la expresión `do`, que realiza un `if` dentro de un bucle infinito. El siguiente programa incrementa o decrementa de forma infinita, pero si encuentra el valor 0 habrá un camino donde se salga del bucle.

```
byte count;

proctype counter() {
    do
    :: count = count + 1
```

```
        :: count = count - 1
        :: (count == 0) -> break
        od
    }
```

Si queremos obligar a que se salga siempre que encuentre un 0, tenemos que escribir el programa de la siguiente manera:

```
    proctype counter() {
        do
        :: (count != 0) ->
            if
            :: count = count + 1
            :: count = count - 1
            fi
        :: (count == 0) -> break
        od
    }
```

4. Ejemplo de especificación Spin/Promela

El siguiente ejemplo modela el problema de la exclusión mutua entre dos procesos asíncronos haciendo uso de un semáforo. Cada proceso tiene cuatro estados: `idle`, `entering`, `critical`, `exiting`. La variable `semaphore` representa el semáforo utilizado y permite que uno de los procesos entre en su parte crítica de ejecución si el semáforo vale `0`, poniendo entonces el semáforo a `1`. Cuando el proceso sale de su parte crítica pone el semáforo de nuevo a `0`. El código del ejemplo es el siguiente:

```
byte mutex=false;
mtype = {idle,entering,critical,exiting}

proctype P(){
mtype state;
 state=idle;
do
  :: state == idle -> state = idle;
  :: state == idle -> state = entering;
  :: (state == entering && mutex==0) ->
        mutex++;
        state = critical;
  :: state == critical ->
        mutex--;
        state = exiting;
  :: state == exiting -> state = idle;
od
}

init{
   atomic { run P(); run P(); }
}

ltl security {[] (mutex < 2)}
```

Si llevamos a cabo su verificación con Spin obtenemos el siguiente resultado, que muestra que el invariante "`mutex` < 2" no se ha satisfecho (hay que prestar atención a la línea con el mensaje "`assertion violated (mutex<2)`"que es la que indica que ha habido un error):

```
$ spin -a mutex1-fail.pml
$ gcc -o pan pan.c
$ ./pan
pan:1: assertion violated (mutex<2) (at depth 39)
pan: wrote mutex1-fail.pml.trail

(Spin Version 6.2.5 -- 3 May 2013)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (none specified)
assertion violations +
acceptance   cycles  - (not selected)
invalid end states +

State-vector 44 byte, depth reached 72, errors: 1
      354 states, stored
      347 states, matched
      701 transitions (= stored+matched)
        2 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.024 equivalent memory usage for states (stored*(State-vector + overhead))
    0.290 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  128.730 total actual memory usage

pan: elapsed time 0 seconds
```

Podemos ver la traza de ejecución del error con el siguiente comando:

```
$ spin -t -p mutex1-fail.pml
using statement merging
Starting P with pid 1
  1: proc  0 (:init:) mutex1-fail.pml:26 (state 1) [(run P(0))]
Starting P with pid 2
  2: proc  0 (:init:) mutex1-fail.pml:26 (state 2) [(run P(1))]
  4: proc  2 (P) mutex1-fail.pml:7 (state 1) [state[i] = idle]
  5: proc  2 (P) mutex1-fail.pml:9 (state 2) [((state[i]==idle))]
  6: proc  1 (P) mutex1-fail.pml:7 (state 1) [state[i] = idle]
  7: proc  2 (P) mutex1-fail.pml:9 (state 3) [state[i] = idle]
  8: proc  2 (P) mutex1-fail.pml:10 (state 4) [((state[i]==idle))]
  9: proc  2 (P) mutex1-fail.pml:10 (state 5) [state[i] = entering]
 10: proc  2 (P) mutex1-fail.pml:11 (state 6) [(((state[i]==entering)&&(mutex==0)))]
 11: proc  2 (P) mutex1-fail.pml:12 (state 7) [mutex = (mutex+1)]
 12: proc  2 (P) mutex1-fail.pml:13 (state 8) [state[i] = critical]
 13: proc  2 (P) mutex1-fail.pml:14 (state 9) [((state[i]==critical))]
```

```
 14: proc  2 (P) mutex1-fail.pml:15 (state 10) [mutex = (mutex-1)]
 15: proc  2 (P) mutex1-fail.pml:16 (state 11) [state[i] = exiting]
 16: proc  2 (P) mutex1-fail.pml:17 (state 12) [((state[i]==exiting))]
 17: proc  1 (P) mutex1-fail.pml:10 (state 4) [((state[i]==idle))]
 18: proc  2 (P) mutex1-fail.pml:17 (state 13) [state[i] = idle]
 19: proc  2 (P) mutex1-fail.pml:9 (state 2) [((state[i]==idle))]
 20: proc  1 (P) mutex1-fail.pml:10 (state 5) [state[i] = entering]
 21: proc  2 (P) mutex1-fail.pml:9 (state 3) [state[i] = idle]
 22: proc  2 (P) mutex1-fail.pml:10 (state 4) [((state[i]==idle))]
 23: proc  2 (P) mutex1-fail.pml:10 (state 5) [state[i] = entering]
 24: proc  2 (P) mutex1-fail.pml:11 (state 6) [(((state[i]==entering)&&(mutex==0)))]
 25: proc  2 (P) mutex1-fail.pml:12 (state 7) [mutex = (mutex+1)]
 26: proc  2 (P) mutex1-fail.pml:13 (state 8) [state[i] = critical]
 27: proc  2 (P) mutex1-fail.pml:14 (state 9) [((state[i]==critical))]
 28: proc  2 (P) mutex1-fail.pml:15 (state 10) [mutex = (mutex-1)]
 29: proc  2 (P) mutex1-fail.pml:16 (state 11) [state[i] = exiting]
 30: proc  2 (P) mutex1-fail.pml:17 (state 12) [((state[i]==exiting))]
 31: proc  1 (P) mutex1-fail.pml:11 (state 6) [(((state[i]==entering)&&(mutex==0)))]
 32: proc  2 (P) mutex1-fail.pml:17 (state 13) [state[i] = idle]
 33: proc  2 (P) mutex1-fail.pml:10 (state 4) [((state[i]==idle))]
 34: proc  2 (P) mutex1-fail.pml:10 (state 5) [state[i] = entering]
 35: proc  2 (P) mutex1-fail.pml:11 (state 6) [(((state[i]==entering)&&(mutex==0)))]
 36: proc  2 (P) mutex1-fail.pml:12 (state 7) [mutex = (mutex+1)]
 37: proc  2 (P) mutex1-fail.pml:13 (state 8) [state[i] = critical]
 38: proc  2 (P) mutex1-fail.pml:14 (state 9) [((state[i]==critical))]
 39: proc  1 (P) mutex1-fail.pml:12 (state 7) [mutex = (mutex+1)]
spin: mutex1-fail.pml:22, Error: assertion violated
spin: text of failed assertion: assert((mutex<2))
spin: trail ends after 40 steps
#processes: 4
mutex = 2
state[0] = entering
state[1] = critical
 40: proc  2 (P) mutex1-fail.pml:15 (state 10)
 40: proc  1 (P) mutex1-fail.pml:13 (state 8)
 40: proc  0 (:init:) mutex1-fail.pml:27 (state 5) <valid end state>
4 processes created
```

En realidad, el problema es que nuestra especificación no es correcta y cada una de las instrucciones de uno de los programas `P` puede ser intercalada con instrucciones del otro programa `P`. La solución consiste en indicar qué operaciones son atómicas, es decir, no pueden ser interrumpidas por instrucciones de otro proceso. La nueva especificación ya solucionada es:

```
byte mutex=false;
mtype = {idle,entering,critical,exiting}

proctype P(){
mtype state;
 state=idle;
do
  :: state == idle -> state = idle;
  :: state == idle -> state = entering;
  :: atomic {
     (state == entering && mutex==0) ->
```

```
        mutex++;
        state = critical;
     }
  :: state == critical ->
     atomic {
        mutex--;
        state = exiting;
     }
  :: state == exiting -> state = idle;
od
}

init{
   atomic { run P(); run P(); }
}

ltl security {[] (mutex < 2)}
```

Si llevamos a cabo su verificación con Spin obtenemos el siguiente resultado, que muestra que el invariante "`mutex` < 2" se cumple (no aparece ninguna expresión de la forma "`assertion violated`"):

```
$ spin -a mutex2.pml
$ gcc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 6.2.5 -- 3 May 2013)
+ Partial Order Reduction

Full statespace search for:
never claim          - (none specified)
assertion violations +
acceptance   cycles  - (not selected)
invalid end states +

State-vector 44 byte, depth reached 48, errors: 0
      232 states, stored
      260 states, matched
      492 transitions (= stored+matched)
        2 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.016 equivalent memory usage for states (stored*(State-vector + overhead))
    0.289 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  128.730 total actual memory usage


unreached in proctype P
mutex2.pml:23, state 19, "-end-"
(1 of 19 states)
unreached in init
```

```
(0 of 5 states)

pan: elapsed time 0 seconds
```

Sin embargo, si queremos verificar las propiedades de vivacidad del ejemplo

```
AG (P.state = entering -> F P.state = critical)
```

como hicimos en la práctica 2 con SMV, dicha propiedad se describe en Spin con las siguientes instrucciones incluidas en el fichero

```
ltl liveness0 {[] ((state[0] == entering) -> (<> (state[0] == critical)))}
ltl liveness1 {[] ((state[1] == entering) -> (<> (state[1] == critical)))}
```

y sabemos que la propiedad falla en este modelo.

La nueva especificación con el estado global de los procesos y la propiedad LTL es el siguiente:

```
byte mutex=false;
mtype = {idle,entering,critical,exiting}
mtype state[2];

proctype P(bit i){
 state[i]=idle;
do
  :: state[i] == idle -> state[i] = idle;
  :: state[i] == idle -> state[i] = entering;
  :: atomic {
     (state[i] == entering && mutex==0) ->
        mutex++;
        state[i] = critical;
     }
  :: state[i] == critical ->
     atomic {
        mutex--;
        state[i] = exiting;
     }
  :: state[i] == exiting -> state[i] = idle;
od
}

init{
   atomic { run P(0); run P(1); }
}

ltl security {[] (mutex < 2)}
ltl liveness0 {[] ((state[0] == entering) -> (<> (state[0] == critical)))}
ltl liveness1 {[] ((state[1] == entering) -> (<> (state[1] == critical)))}
```

Si llevamos a cabo su verificación (usando el parámetro "`-a`" para verificar *acceptance cycles*, es decir, ciclos indeseables y el parámetro "`-f`" para incluir restricciones de fairness) con Spin obtenemos el siguiente resultado:

```
$ ./spin -a mutex3 -N liveness0
liveness0.pml
```

```
ltl liveness0: [] ((! ((state[0]==entering))) || ((state[0]==critical)))
$ gcc -o pan pan.c
$ ./pan -a -f -N liveness0
pan:1: assertion violated  !( !(( !((state[0]==3))||(333==2)))) (at depth 38)
pan: wrote liveness0.trail

(Spin Version 6.2.5 -- 3 May 2013)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          + (liveness)
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness enabled)
invalid end states - (disabled by never claim)

State-vector 52 byte, depth reached 45, errors: 1
       27 states, stored
       14 states, matched
       41 transitions (= stored+matched)
        2 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.002 equivalent memory usage for states (stored*(State-vector + overhead))
    0.288 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  128.730 total actual memory usage

pan: elapsed time 0.01 seconds
```

Podemos ver la traza de ejecución del error con el siguiente comando:

```
$ ./spin -t -p mutex3 -N liveness0
liveness0.pml
ltl liveness0: [] ((! ((state[0]==entering))) || ((333==critical)))
starting claim 3
using statement merging
Never claim moves to line 4 [(1)]
  2: proc  0 (:init:) liveness0:30 (state 1) [mutex = 0]
Starting P with pid 2
  4: proc  0 (:init:) liveness0:31 (state 2) [(run P(0))]
Starting P with pid 3
  5: proc  0 (:init:) liveness0:31 (state 3) [(run P(1))]
 10: proc 3 terminates
 12: proc  2 (P) liveness0:7 (state 1) [state[i] = idle]
 14: proc  2 (P) liveness0:9 (state 2) [((state[i]==idle))]
 16: proc  1 (P) liveness0:7 (state 1) [state[i] = idle]
 18: proc  2 (P) liveness0:9 (state 3) [state[i] = idle]
 20: proc  2 (P) liveness0:10 (state 4) [((state[i]==idle))]
 22: proc  2 (P) liveness0:10 (state 5) [state[i] = entering]
 24: proc  2 (P) liveness0:12 (state 6) [(((state[i]==entering)&&(mutex==0)))]
 24: proc  2 (P) liveness0:13 (state 7) [mutex = (mutex+1)]
 24: proc  2 (P) liveness0:14 (state 8) [state[i] = critical]
```

```
 26: proc  2 (P) liveness0:16 (state 10) [((state[i]==critical))]
 28: proc  2 (P) liveness0:18 (state 11) [mutex = (mutex-1)]
 28: proc  2 (P) liveness0:19 (state 12) [state[i] = exiting]
 30: proc  2 (P) liveness0:21 (state 14) [((state[i]==exiting))]
 32: proc  1 (P) liveness0:10 (state 4) [((state[i]==idle))]
 34: proc  2 (P) liveness0:21 (state 15) [state[i] = idle]
 36: proc  2 (P) liveness0:9 (state 2) [((state[i]==idle))]
 38: proc  1 (P) liveness0:10 (state 5) [state[i] = entering]
spin: _spin_nvr.tmp:3, Error: assertion violated
spin: text of failed assertion: assert(!(!(((!((state[0]==entering))||(333==critical)))))
Never claim moves to line 3 [assert(!(!(((!((state[0]==entering))||(333==critical)))))]
spin: trail ends after 39 steps
#processes: 3
mutex = 0
state[0] = entering
state[1] = idle
 39: proc  2 (P) liveness0:9 (state 3)
 39: proc  1 (P) liveness0:8 (state 16)
 39: proc  0 (:init:) liveness0:32 (state 6) <valid end state>
 39: proc  - (liveness) _spin_nvr.tmp:2 (state 6)
4 processes created
```

Esta traza muestra el mismo problema que en la práctica anterior con SMV.

5. Objetivo de la práctica

El objetivo de esta práctica consiste en realizar la especificación de los siguientes problemas en Spin y comprobar algunas propiedades usando la lógica temporal LTL.

Evaluación: La evaluación de prácticas se realizará con un examen tipo test en la fecha concretada. Antes del día del examen hay que enseñarle al profesor los ejercicios realizados como parte de la nota de seguimiento.

5.1. Solucionar propiedad de vivacidad

Realizar las modificaciones necesarias al ejemplo de introducción para que cumpla las dos propiedades antes mencionadas. Para ello deberéis considerar una variable que indique quién tiene el turno para acceder a su zona crítica, obligando así a respetar ese turno.

El código a considerar como base es el siguiente:

```
bit turn=0;
byte mutex=false;
mtype = {idle,entering,critical,exiting}
mtype state[2];

proctype P(bit i){
 state[i]=idle;
do
  :: state[i] == idle -> state[i] = idle;
  :: state[i] == idle -> state[i] = entering;
  :: atomic {
     (state[i] == entering && mutex==0) ->
        mutex++;
```

```
        state[i] = critical;
     }
  :: state[i] == critical ->
     atomic {
        mutex--;
        state[i] = exiting;
     }
  :: state[i] == exiting -> state[i] = idle;
od
}

init{
   atomic { run P(0); run P(1); }
}

ltl security {[] (mutex < 2)}
ltl liveness0 {[] ((state[0] == entering) -> (<> (state[0] == critical)))}
ltl liveness1 {[] ((state[1] == entering) -> (<> (state[1] == critical)))}
```

5.2. Un ascensor

El problema consiste en modelar el comportamiento de un ascensor de 4 plantas. Cada una de las plantas tiene un botón de llamada. Además, el ascensor tiene un botón de planta para cada una de las plantas en su interior, que los pasajeros presionarán para ir a una determinada planta. El recorrido del ascensor de una planta a otra durará una única unidad de tiempo, es decir, el cambio de la planta 2 a la planta 3 debe realizarse en un único cambio de estado. Cuando llega a una de las plantas, el ascensor abre las puertas y las mantiene abiertas durante una unidad de tiempo, luego las cierra y procede a servir otra petición. El ascensor deberá mantener una dirección hasta satisfacer todas las peticiones existentes en esa dirección.

Deberéis especificar el problema como un programa que satisfaga las siguientes propiedades: cada llamada al ascensor desde una planta es eventualmente satisfecha, la pulsación de un botón de planta es eventualmente satisfecha y, finalmente, el ascensor nunca se moverá con las puertas abiertas.

Nota: La siguiente plantilla puede ayudar.

```
bool botones[4];
int planta = 0;
mtype = { arriba, parado, abajo, abiertas, cerradas };
mtype puertas = cerradas;
mtype direccion = parado;

init{
  botones[0] = false;
  botones[1] = true;
  botones[2] = false;
  botones[3] = false;
  do
    ::puertas == cerradas && direccion == parado && botones[planta] ->
        puertas = abiertas;
        botones[planta] = false;
    ::planta == 1 && direccion == arriba && (botones[2] || botones[3])
       && puertas == cerradas ->
        planta = planta + 1;
```

```
    ...
  od;

}
```

5.3. Alternating-bit-protocol

El protocolo de bit de alternancia es un protocolo muy simple pero efectivo que evita la aparición de mensajes duplicados o perdidos. Considérese un emisor A, un receptor B y un canal de A a B, que suponemos que inicialmente no transmite ningún mensaje. Cada mensaje contiene un bit de protocolo (valores 0 o 1).

Cuando A tiene un mensaje para enviar a B, lo repite indefinidamente por el canal de A a B utilizando el mismo bit de protocolo hasta que recibe una confirmación (acknowledgement) de B con el mismo bit de protocolo. Al recibir la confirmación, A empieza a transmitir el siguiente mensaje disponible con el bit de protocolo cambiado, es decir, mensaje anterior: 0, siguiente mensaje: 1 o al revés. Cuando B recibe un mensaje, lo acepta para procesar y manda a A una confirmación incluyendo el mismo bit de protocolo del mensaje recibido. Ante sucesivos mensajes con el mismo bit de protocolo, lo único que hace es mandar la misma confirmación con el mismo bit de protocolo.

Es decir, básicamente, el bit de protocolo identifica cada uno de los mensajes y el emisor repite el mensaje hasta que recibe una confirmación del receptor, a su vez el receptor repite siempre la confirmación del mensaje que le llega.

La propiedad a verificar es que siempre que no se haya perdido los datos del canal del emisor al receptor y que no se hayan perdido los datos del canal del receptor al emisor, la confirmación recibida por el emisor corresponde justamente al dato que estaba enviando.

Nota: La siguiente plantilla puede ayudar.

```
mtype = { vacio, cero, uno };
mtype enviaA = cero;
mtype enviaB = vacio;
mtype recibeA = vacio;
mtype recibeB = vacio;

proctype sender() {
  do
    ::atomic
      { (recibeA == enviaA && recibeA != vacio & enviaA==cero) -> enviaA = uno }
    ::atomic
      { (recibeA == enviaA && recibeA != vacio & enviaA==uno) -> enviaA = cero }
  od
}

proctype receiver() {
  do
    ...
  od
}

proctype channel(mtype input; mtype output) {
  do
    :: output = input;
    :: output = vacio;
  od
```

```
}

init {atomic {run sender(); run receiver(); run channel(..., ...); run channel(..., ...); }}
```