Manual de introducción a Spin

Adrián Riesco

Especificación, Validación y Testing

Curso 2014/15

Índice

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 solo 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 Lógica Lineal Temporal (LTL por sus siglas en inglés). 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 otros lenguajes como 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 señalización en vías de trenes, etc. La NASA lo utiliza de forma rutinaria para verificar

software, e.g. en las misiones Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc.

2. Lógica Lineal Temporal

La *lógica lineal temporal* (LTL por sus siglas en inglés) 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

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 y A es una expresión temporal. Fijando un estado actual y un camino a partir de él, determinamos la satisfactibilidad de una fórmula de camino p con respecto a los estados siguientes al actual dentro del camino fijado.

- Fórmula $\bigcirc\ p$ (Next): la proposición p debe satisfacerse en el estado siguiente al actual.
- Fórmula $\Box\ p$ (Globally): la proposición p debe satisfacerse en todos los estados posteriores al estado actual.
- Fórmula $\Diamond\ p$ (Eventually): la proposición p debe satisfacerse en al menos un estado posterior al estado actual.
- 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. $\Box\ p$ indica que siempre llueve.
2. $\Diamond\ p$ indica que siempre llegará un momento en que llueva.
3. $\Diamond\ \Box\ p$ indica que siempre llegará un momento en que llueva y no pare de llover.
4. $\Box\ \Diamond\ (!p \wedge q)$ indica que siempre llegará un momento en que no llueve y juegue al tenis y esto ocurrirá un número infinito de veces.
5. $\bigcirc\ p$ indica que va a llover en el siguiente instante de tiempo, pase lo que pase.
6. $q\ U\ p$ indica que juego al tenis hasta que empieza a llover.

3. El sistema Spin

El sistema Spin comprueba que un sistema software distribuido satisface la especificación asociada proporcionada en 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 correción 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.

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 continuá 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.

En los laboratorios usaremos la interfaz gráfica desarrollada para Spin, llamada iSpin. Dicha interfaz se invoca usando el comando `ispin` desde la línea de comandos.

4. 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	$0 \ldots 255$
chan	$1 \ldots 255$
mtype	$1 \ldots 255$
pid	$0 \ldots 255$
short	$-2^{15} \ldots 2^{15} - 1$
int	$-2^{31} \ldots 2^{31} - 1$
unsigned	$0 \ldots 2^{n} - 1$

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` solo 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 este curso no vamos a describir los canales de comunicación ya que no son necesarios para hacer las prácticas 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 solo 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 aquellas selecciones que no estén bloquedas. En el ejemplo anterior las dos condiciones son excluyentes y el `if` escogerá solo 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 sí 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
}
```

5. Ejemplo de especificación en 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` y `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 sección crítica pone el semáforo de nuevo a 0. El código del ejemplo, disponible en el Campus Virtual en el fichero `semaforo.pml`, 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
  }

  proctype monitor(){
    assert(mutex<2);
  }

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

Una vez cargado el fichero en iSpin con el botón `Open`, podemos pasar a la pestaña `Verification` y, sin cambiar las opciones por defecto, podemos usar el botón `Run` para comprobar si la propiedad que hemos indicado en el proceso `monitor` se cumple. Al llevar a cabo la 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 6, en la que aparece el mensaje "`assertion violated (mutex<2)`" que es la que indica que ha habido un error):

```
verification result:
spin -a  semaforo.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 50356
pan:1: assertion violated (mutex<2) (at depth 30)
pan: wrote semaforo.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (not selected)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 36 byte, depth reached 46, errors: 1
      108 states, stored
       73 states, matched
      181 transitions (= stored+matched)
        2 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.007 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 del programa si pasamos a la pestaña `Simulate/Replay` y pulsamos el botón `(Re)Run`:

```
Starting P with pid 1
  1: proc  0 (:init::1) semaforo.pml:28 (state 1) [(run P())]
Starting P with pid 2
  2: proc  0 (:init::1) semaforo.pml:28 (state 2) [(run P())]
Starting monitor with pid 3
  3: proc  0 (:init::1) semaforo.pml:28 (state 3) [(run monitor())]
  4: proc  2 (P:1) semaforo.pml:8 (state 1) [state = idle]
  5: proc  1 (P:1) semaforo.pml:8 (state 1) [state = idle]
  6: proc  2 (P:1) semaforo.pml:11 (state 4) [((state==idle))]
  6: proc  2 (P:1) semaforo.pml:11 (state 5) [state = entering]
  7: proc  2 (P:1) semaforo.pml:12 (state 6) [(((state==entering)&&(mutex==0)))]
  8: proc  2 (P:1) semaforo.pml:12 (state 7) [mutex = (mutex+1)]
  9: proc  2 (P:1) semaforo.pml:13 (state 8) [state = critical]
 10: proc  2 (P:1) semaforo.pml:14 (state 9) [((state==critical))]
 11: proc  2 (P:1) semaforo.pml:14 (state 10) [mutex = (mutex-1)]
 12: proc  2 (P:1) semaforo.pml:15 (state 11) [state = exiting]
 13: proc  1 (P:1) semaforo.pml:11 (state 4) [((state==idle))]
 13: proc  1 (P:1) semaforo.pml:11 (state 5) [state = entering]
 14: proc  2 (P:1) semaforo.pml:16 (state 12) [((state==exiting))]
 14: proc  2 (P:1) semaforo.pml:16 (state 13) [state = idle]
 15: proc  2 (P:1) semaforo.pml:11 (state 4) [((state==idle))]
 15: proc  2 (P:1) semaforo.pml:11 (state 5) [state = entering]
 16: proc  2 (P:1) semaforo.pml:12 (state 6) [(((state==entering)&&(mutex==0)))]
 17: proc  2 (P:1) semaforo.pml:12 (state 7) [mutex = (mutex+1)]
 18: proc  2 (P:1) semaforo.pml:13 (state 8) [state = critical]
 19: proc  2 (P:1) semaforo.pml:14 (state 9) [((state==critical))]
 20: proc  2 (P:1) semaforo.pml:14 (state 10) [mutex = (mutex-1)]
 21: proc  1 (P:1) semaforo.pml:12 (state 6) [(((state==entering)&&(mutex==0)))]
 22: proc  2 (P:1) semaforo.pml:15 (state 11) [state = exiting]
 23: proc  2 (P:1) semaforo.pml:16 (state 12) [((state==exiting))]
 23: proc  2 (P:1) semaforo.pml:16 (state 13) [state = idle]
 24: proc  2 (P:1) semaforo.pml:11 (state 4) [((state==idle))]
 24: proc  2 (P:1) semaforo.pml:11 (state 5) [state = entering]
 25: proc  2 (P:1) semaforo.pml:12 (state 6) [(((state==entering)&&(mutex==0)))]
 26: proc  2 (P:1) semaforo.pml:12 (state 7) [mutex = (mutex+1)]
 27: proc  2 (P:1) semaforo.pml:13 (state 8) [state = critical]
 28: proc  2 (P:1) semaforo.pml:14 (state 9) [((state==critical))]
 29: proc  1 (P:1) semaforo.pml:12 (state 7) [mutex = (mutex+1)]
 30: proc  1 (P:1) semaforo.pml:13 (state 8) [state = critical]
spin: semaforo.pml:22, Error: assertion violated
spin: text of failed assertion: assert((mutex<2))
#processes: 4
 31: proc  3 (monitor:1) semaforo.pml:22 (state 1)
 31: proc  2 (P:1) semaforo.pml:14 (state 10)
 31: proc  1 (P:1) semaforo.pml:9 (state 14)
 31: proc  0 (:init::1) semaforo.pml:29 (state 5)
4 processes created
Exit-Status 0
```

Usando el botón `Rewind` podemos ejecutar el programa y navegar por la traza hacia delante y detrás con los botones `Step Forward` y `Step Backward`, respectivamente. 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
}

proctype monitor(){
  assert(mutex<2);
}

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

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

```
verification result:
spin -a  semaforo_fixed.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 69717

(Spin Version 6.3.2 -- 17 May 2014)
+ Partial Order Reduction

Full statespace search for:
never claim          - (not selected)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 36 byte, depth reached 21, errors: 0
       66 states, stored
       76 states, matched
      142 transitions (= stored+matched)
```

```
        2 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.004 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
semaforo_fixed.pml:20, state 19, "-end-"
(1 of 19 states)
unreached in proctype monitor
(0 of 2 states)
unreached in init
(0 of 5 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```