

Ingeniería del Software II

6 - Model Checking

El problema de model checking

Dado un modelo M de un sistema (en algún lenguaje) y una propiedad ϕ (en alguna lógica) deseamos verificar automáticamente si esta es satisfecha por M , es decir, si $M \models \phi$.

En particular para el caso de LTL:

- sabemos que el lenguaje $\mathcal{L}(\phi)$ de una fórmula es el conjunto de todas las trazas donde ésta se hace verdadera, y que
- el comportamiento de un sistema M (denotado $\mathcal{L}(M)$) está dado por el conjunto de todas las trazas que éste puede ejecutar.

Luego, $M \models \phi$ si y sólo si toda traza de M satisface ϕ , es decir:

$$M \models \phi \text{ si y sólo si } \mathcal{L}(M) \subseteq \mathcal{L}(\phi)$$

El problema de model checking se reduce entonces a validar esta inclusión de manera automática

¿Pero cómo?

Cómo obtener el modelo de un sistema

Para poder definir $\mathcal{L}(M)$ necesitamos primero una manera razonable de definir M . Pero eso ya sabemos como hacerlo:

```


int y1 = 0;
int y2 = 0;
short in_critical = 0;

active proctype process_1() {
    do
        :: true ->
0:         y1 = y2+1;
1:         ((y2==0) || (y1<=y2));
           in_critical++;
2:         in_critical--;
3:         y1 = 0;
    od
}
```

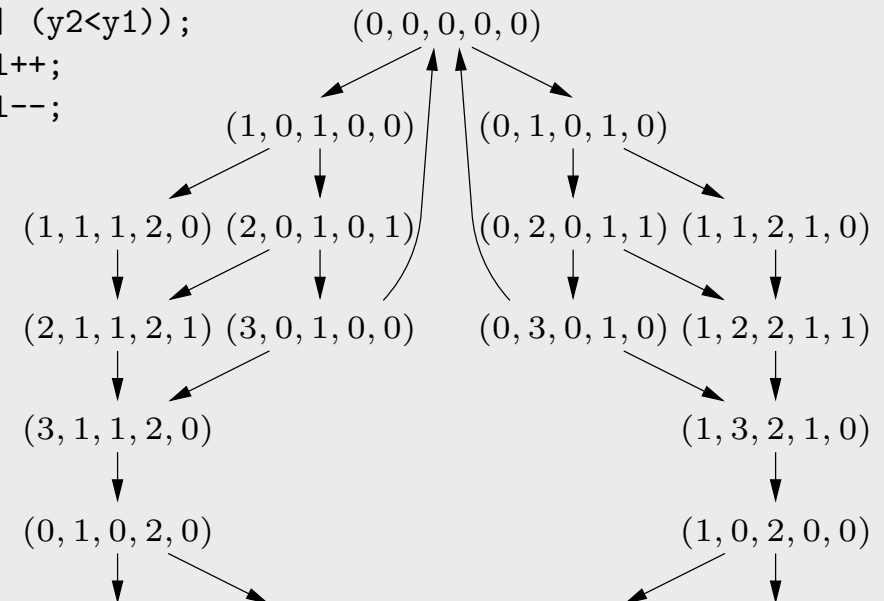
```

active proctype process_2() {
    do
        :: true ->
0:         y2 = y1+1;
1:         ((y1==0) || (y2<y1));
           in_critical++;
2:         in_critical--;
3:         y2 = 0;
    od
}

```

(1, 1, 1, 2, 0) 

Estructura del estado:

$$(pc_1, pc_2, y1, y2, \text{in_critical})$$


Cómo obtener el modelo de un sistema

Es decir, el modelo del sistema define un sistema de transiciones.

Un **sistema de transiciones** es una estructura

$$M = (S, s_0, \rightarrow, v)$$

donde:

- S es un conjunto de **estados** donde $s_0 \in S$ es el **estado inicial**,
- $\rightarrow \subseteq S \times S$ es la **relación de transición** tal que $\forall s \in S : \exists s' \in S : s \rightarrow s'$
- $v : S \rightarrow 2^{PA}$ es una función de **valuación**.

$v(s)$ es el conjunto de todas las proposiciones atómicas que son verdaderas en el estado s .

(En el contexto de lógicas modales, esta estructura se denomina estructura de Kripke)

Cómo obtener el modelo de un sistema

Una ejecución de M es una función $\rho : \mathbb{N} \rightarrow S$ tal que:

1. $\rho(0) = s_0$, y
2. $\rho(i) \rightarrow \rho(i+1)$ para todo $i \geq 0$

El comportamiento de M se define como:

$$\mathcal{L}(M) = \{\sigma \in (2^{PA})^\omega \mid \exists \rho \text{ ejecución de } M : \\ \forall i \geq 0 : \sigma(i) = v(\rho(i))\}$$

Autómatas de Büchi

Al igual que los lenguajes regulares, los lenguajes ω -regulares no son fáciles de manipular por sí mismos.

De la misma manera que los lenguajes regulares se manipulan a través de autómatas finitos que acepten los lenguajes a manipular, los lenguajes ω -regulares pueden manipularse a través de los denominados autómatas de Büchi.

Un autómata de Büchi es una estructura

$$\mathcal{A} = (\Sigma, S, \delta, s_0, A)$$

en la cual:

- Σ es un conjunto finito, llamado **alfabeto**,
- S es un conjunto finito de **estados** donde $s_0 \in S$ es el **estado inicial**,
- $\delta : \Sigma \times S \rightarrow 2^S$ es la función de **transición**,
- A es el conjunto de **estados de aceptación**.

Aceptación de trazas en autómatas de Büchi

Dado un autómata de Buchi \mathcal{A} , decimos que una traza $\sigma = \sigma_0 \sigma_1 \sigma_2 \dots$ de elementos de Σ es **aceptada** por \mathcal{A} sii existe una función $\rho : \mathbb{N} \rightarrow S$ (denominada ejecución) tal que:

- $\rho(0) = s_0$,
- $\rho(i + 1) \in \delta(\sigma(i), \rho(i))$ para todo $i \geq 0$, y
- existen estados de aceptación en A que se repiten infinitas veces en ρ , i.e., el conjunto $\{i \in \mathbb{N} \mid \rho(i) \in A\}$ es infinito.

Definimos como el **lenguaje de \mathcal{A}** , notación $\mathcal{L}(\mathcal{A})$, al conjunto de todas las trazas (i.e., ω -palabras) aceptadas por \mathcal{A} .

Los autómatas de Büchi aceptan exactamente todos los lenguajes ω -regulares. Por consiguiente son más expresivos que LTL sobre el alfabeto 2^{PA} .

Autómatas de Büchi como modelos de sistemas

Ya dijimos que un programa P cuyo espacio de estado sea finito puede representarse con un sistema de transiciones finito $M_P = (S, s_0, \rightarrow, v)$.

A su vez, M_P puede verse como el autómata de Büchi $\mathcal{A}_P = (\Sigma, S, \delta, s_0, S)$, donde:

- $\Sigma = 2^{PA}$,
- $s_j \in \delta(B, s_i)$ sii $s_i \rightarrow s_j \wedge B = v(s_i)$

Notar que **todos** los estados son estados de aceptación. Esto es así porque nos interesan todas las ejecuciones posibles del sistema.

Teorema: $\mathcal{L}(M_P) = \mathcal{L}(\mathcal{A}_P)$

Autómatas de Büchi como modelos de sistemas

Ya dijimos que un programa P cuyo espacio de estado sea finito puede representarse con un sistema de transiciones finito $M_P = (S, s_0, \rightarrow, v)$.

A su vez, M_P puede verse como el autómata de Büchi $\mathcal{A}_P = (\Sigma, S, \delta, s_0, S)$, donde:

- $\Sigma = 2^{PA}$,
- $s_j \in \delta(B, s_i)$ sii $s_i \rightarrow s_j \wedge B = v(s_i)$

Notar que **todos** los estados son estados de aceptación. Esto es así porque nos interesan todas las posibles del sistema.

Demostrar formalmente este teorema.

Teorema: $\mathcal{L}(M_P) = \mathcal{L}(\mathcal{A}_P)$

Fórmulas LTL y Autómatas de Büchi

Teorema: Para toda fórmula LTL ϕ , se puede construir un autómata de Büchi \mathcal{A}_ϕ tal que:

$$\mathcal{L}(\mathcal{A}_\phi) = \mathcal{L}(\phi)$$

La demostración de este teorema es compleja. Para formarse una idea de lo establecido por el teorema daremos algunos ejemplos:

$$\diamond p$$

$$\square p$$

$$p \cup q$$

Fórmulas LTL y Autómatas de Büchi

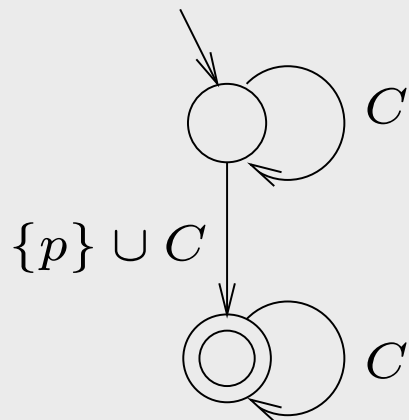
Teorema: Para toda fórmula LTL ϕ , se puede construir un autómatata de Büchi \mathcal{A}_ϕ tal que:

$$\mathcal{L}(\mathcal{A}_\phi) = \mathcal{L}(\phi)$$

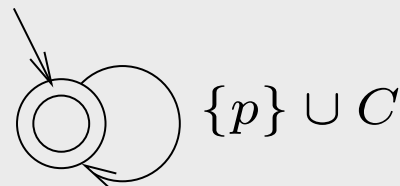
La demostración de este teorema se basa en la idea de lo establecido por el teorema de equivalencia entre PA y LTL.

C es cualquier subconjunto de \mathcal{PA} .
Es decir, cada flecha de los dibujos representa muchas transiciones a la vez.

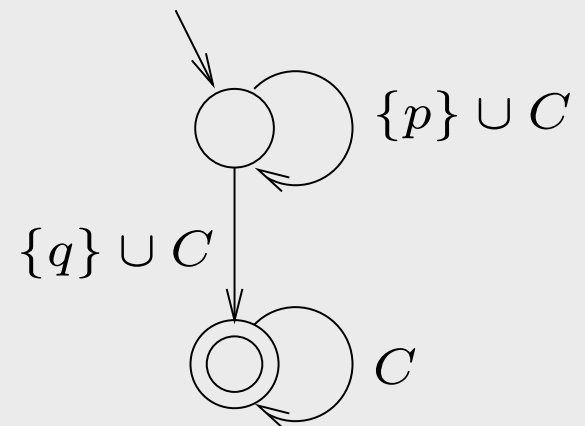
$\diamond p$



$\square p$



$p \cup q$



Manipulación de lenguajes ω -regulares usando autómatas de Büchi

Teorema: Dados dos autómatas de Büchi \mathcal{A}_1 y \mathcal{A}_2 , se puede construir un automata $\mathcal{A}_1 \cap \mathcal{A}_2$ tal que:

$$\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$$

Teorema: Dado un autómadada de Büchi \mathcal{A} se puede construir un automata \mathcal{A}^c tal que:

$$\mathcal{L}(\mathcal{A}^c) = \overline{\mathcal{L}(\mathcal{A})}$$

Teorema: Existe un algoritmo que permite decidir si el lenguaje ω -regular aceptado por un autómata de Büchi es vacío o no.

Manipulación de lenguajes ω -regulares usando autómatas de Büchi

Teorema: Dados dos autómatas de Büchi \mathcal{A}_1 y \mathcal{A}_2 , se puede construir un automata $\mathcal{A}_1 \cap \mathcal{A}_2$ tal que:

$$\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$$

Teorema: Dado un autómata de Büchi \mathcal{A} , se puede construir un automata \mathcal{A}^c tal que:

$$\mathcal{L}(\mathcal{A}^c) = \overline{\mathcal{L}(\mathcal{A})}$$

El algoritmo es un doble DFS con el fin de buscar componentes fuertemente conexas que atrapen un estado de aceptación.

Teorema: Existe un algoritmo que permite decidir si el lenguaje ω -regular aceptado por un autómata de Büchi es vacío o no.

Model checking con fundamentos en la teoría de autómatas

Hemos visto que la verificación de que un programa P de estados finitos satisfaga una propiedad temporal ϕ (i.e., $M_P \models \phi$) se reduce a comprobar que

$$\mathcal{L}(M_P) \subseteq \mathcal{L}(\phi)$$

Con los resultados anteriores, podemos reducir este problema a verificar si

$$\mathcal{L}(\mathcal{A}_P) \subseteq \mathcal{L}(\mathcal{A}_\phi)$$

que a su vez es equivalente a verificar si:

$$\mathcal{L}(\mathcal{A}_P) \cap \overline{\mathcal{L}(\mathcal{A}_\phi)} = \emptyset$$

Problema: Complementar un autómata de Büchi es computacionalmente muy caro (se produce una explosión exponencial).



¿Cómo podemos
evitar este problema?

Model checking con fundamentos en la teoría de autómatas

Hemos visto que la verificación de que un programa P de estados finitos satisfaga una propiedad temporal ϕ (i.e., $M_P \models \phi$) se reduce a comprobar que

$$\mathcal{L}(M_P) \subseteq \mathcal{L}(\phi)$$

Con los resultados anteriores, podemos reducir este problema a verificar si

$$\mathcal{L}(\mathcal{A}_P) \subseteq \mathcal{L}(\mathcal{A}_\phi)$$

que a su vez es equivalente a verificar si:

$$\mathcal{L}(\mathcal{A}_P) \cap \overline{\mathcal{L}(\mathcal{A}_\phi)} = \emptyset$$

Problema: Complementar un autómata de Büchi es computacionalmente muy caro (se produce una explosión exponencial). Por suerte:

$$\overline{\mathcal{L}(\mathcal{A}_\phi)} = \mathcal{L}(\mathcal{A}_{\neg\phi})$$

Por lo tanto, podemos verificar equivalentemente que:

$$\mathcal{L}(\mathcal{A}_P) \cap \mathcal{L}(\mathcal{A}_{\neg\phi}) = \emptyset$$

El algoritmo de model checking “in a nutshell”

Aplicando los resultados anteriores el problema de verificar si un programa P satisface una propiedad ϕ puede esquematizarse como sigue:

1. Construir el autómata de Büchi \mathcal{A}_P
2. Construir el autómata de Büchi $\mathcal{A}_{\neg\phi}$
3. Construir el autómata de Büchi $\mathcal{A}_P \cap \mathcal{A}_{\neg\phi}$
4. Comprobar si $\mathcal{L}(\mathcal{A}_P \cap \mathcal{A}_{\neg\phi})$ es vacío

Un aspecto muy importante del model checking (sino el más importante) es la obtención de un contraejemplo en caso de que la propiedad no sea verdadera:
¿Cómo se obtiene tal contraejemplo?

$$i \mathcal{M} \models \phi?$$

```
int y1 = 0;
int y2 = 0;
short in_critical = 0;
```

 \mathcal{M}

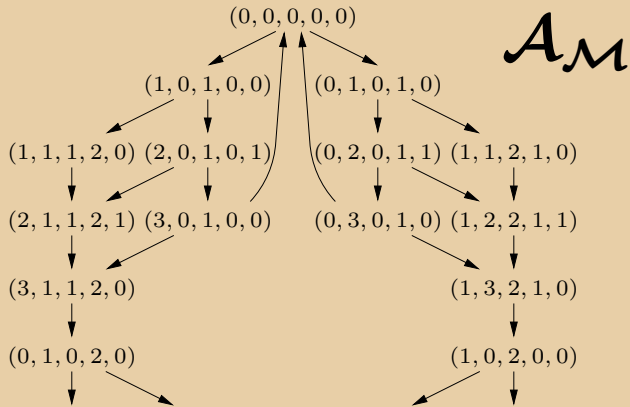
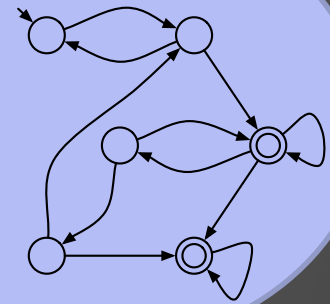
```
active proctype process_1() {
  do
    :: true ->
0:   y1 = y2+1;
1:   ((y2==0) || (y1<=y2));
    in_critical++;
2:   in_critical--;
3:   y1 = 0;
  od
}

active proctype process_2() {
  do
    :: true ->
0:   y2 = y1+1;
1:   ((y1==0) || (y2<y1));
    in_critical++;
2:   in_critical--;
3:   y2 = 0;
  od
}
```

$$\phi : \square \diamond crit_1 \wedge \square \diamond crit_2$$

$$\neg \phi : \neg(\square \diamond crit_1 \wedge \square \diamond crit_2)$$

El problema de
model checking
(para LTL)


 $\mathcal{A}_{\neg \phi}$


$$i \mathcal{A}_{\mathcal{M}} \cap \mathcal{A}_{\neg \phi} = \emptyset?$$

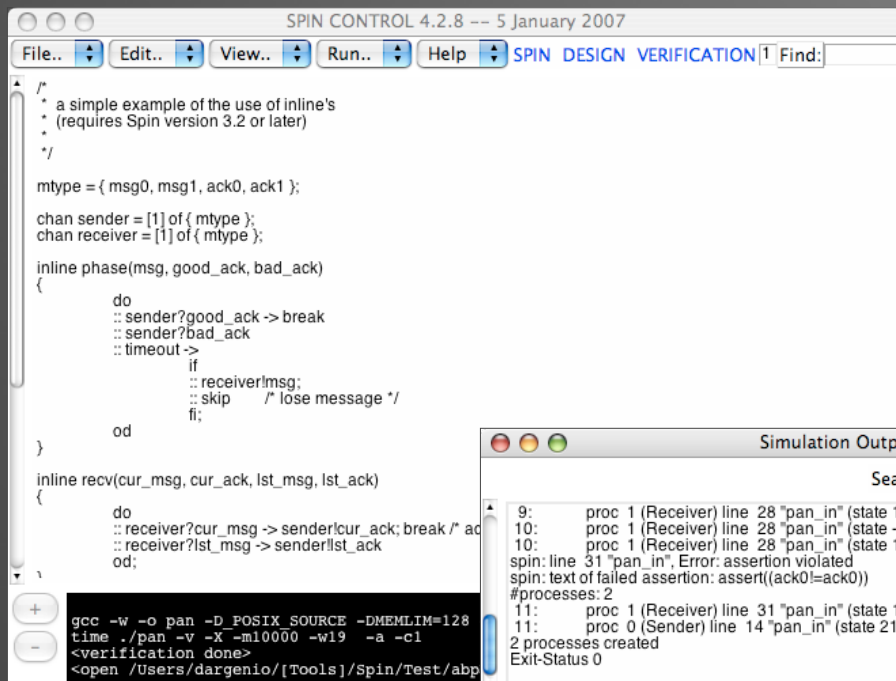
Model Checking: características

- Además de determinar si una propiedad se cumple, dan **contraejemplos** en caso de que no se cumpla.
- El algoritmo básico se basa en “**fuerza bruta**”: recorre todo el grafo subyacente.
- Esto se agrava con el problema de la **explosión de estados**.
 - Se agranda **exponencialmente** con cada variable y cada proceso.
- El grafo subyacente usualmente necesita ser **finito**.

Herramientas de Model Checking

El model checker SPIN

- Desarrollado en AT&T / Bell Labs.
- Principalmente desarrollado por Gerard Holzmann
- Bibliografía:
 - G. Holzmann. The Spin Model Checker. Addisson Wesley. 2004.
- www.spinroot.com

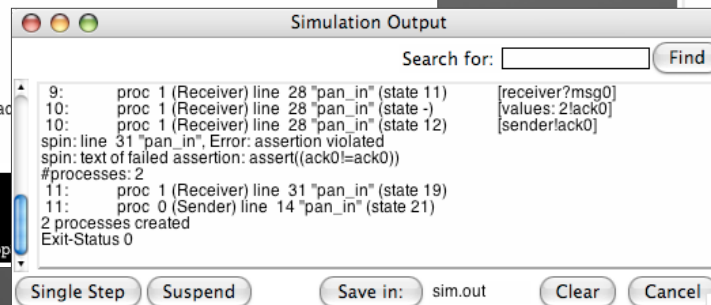


SPIN CONTROL 4.2.8 -- 5 January 2007

```
File.. Edit.. View.. Run.. Help SPIN DESIGN VERIFICATION Find:
```

```
/*  
 * a simple example of the use of inline's  
 * (requires Spin version 3.2 or later)  
 */  
  
mtype = { msg0, msg1, ack0, ack1 };  
chan sender = [1] of { mtype };  
chan receiver = [1] of { mtype };  
  
inline phase(msg, good_ack, bad_ack)  
{  
    do  
        :: sender?good_ack -> break  
        :: sender?bad_ack  
        :: timeout ->  
            if  
                :: receiver!msg;  
                :: skip /* lose message */  
            fi;  
    od  
}  
  
inline rcv(cur_msg, cur_ack, lst_msg, lst_ack)  
{  
    do  
        :: receiver?cur_msg -> sender!cur_ack; break /* ok */  
        :: receiver?lst_msg -> sender!lst_ack  
    od;  
}
```

gcc -w -o pan -D POSIX_SOURCE -DMEMLIM=128
time ./pan -v -X -m10000 -w19 -a -c1
<verification done>
<open /Users/dargenio/[Tools]/Spin/Test/abp

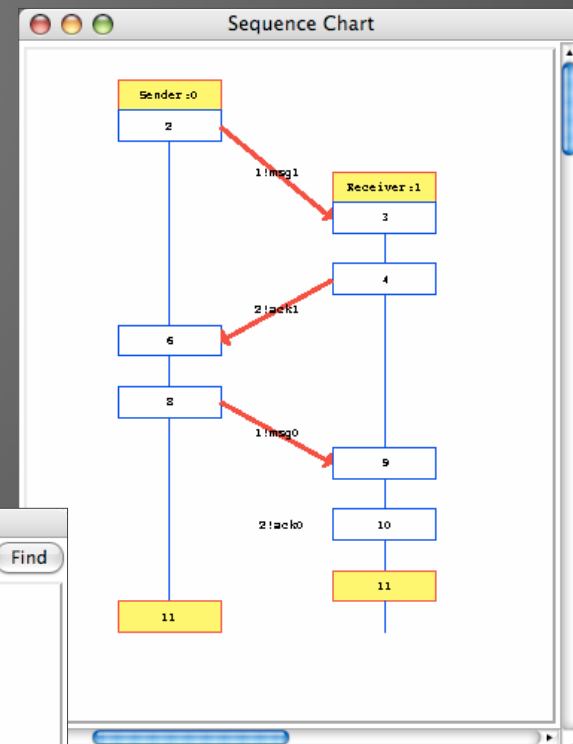


Simulation Output

Search for: Find

```
9: proc 1 (Receiver) line 28 "pan_in" (state 11) [receiver?msg0]  
10: proc 1 (Receiver) line 28 "pan_in" (state -) [values: 2!ack0]  
10: proc 1 (Receiver) line 28 "pan_in" (state 12) [sender!ack0]  
spin: line 31 "pan_in": Error: assertion violated  
spin: text of failed assertion: assert((ack0!=ack0))  
#processes: 2  
11: proc 1 (Receiver) line 31 "pan_in" (state 19)  
11: proc 0 (Sender) line 14 "pan_in" (state 21)  
2 processes created  
Exit-Status 0
```

Single Step Suspend Save in: sim.out Clear Cancel



- La descripción de los modelos se realiza en PROMELA
- Promela se asemeja a C y agrega primitivas para manejar concurrencia, canales, atomicidad, no determinismo, ...

```

/*
 * The alternating bit protocol.
 * A simple example of the use of inline's
 */

```

```

mtype = { msg0, msg1, ack0, ack1 };

```

```

chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };

```

```

inline phase(msg, good_ack, bad_ack)
{

```

```

    do
        :: sender?good_ack -> break
        :: sender?bad_ack
        :: timeout ->
            if
                :: receiver!msg;
                :: skip/* lose message */
            fi;
    od
}

```

```

inline recv(cur_msg, cur_ack, lst_msg, lst_ack)
{
    do
        :: receiver?cur_msg -> sender!cur_ack;
                                break /* accept */
        :: receiver?lst_msg -> sender!lst_ack
    od;
}

```

```

active proctype Sender()
{
    do
        :: phase(msg1, ack1, ack0);
        phase(msg0, ack0, ack1)
    od
}

```

```

active proctype Receiver()
{
    do
        :: recv(msg1, ack1, msg0, ack0);
        recv(msg0, ack0, msg1, ack1)
    od
}

```

Permite realizar simulaciones guiadas, aleatorias, sobre una traza específica (ej: contraejemplo de una propiedad).

The screenshot displays the SPIN CONTROL 4.2.8 interface, which includes a code editor, a sequence chart, and a simulation output window.

Code Editor: The main window shows the SPIN source code for a receiver process. The code defines a channel `receiver` and a `phase` function. The `phase` function handles incoming messages and sends acknowledgments. The `recv` function is also defined. The code is as follows:

```
chan receiver = [1] of { mtype };

inline phase(msg, good_ack, bad_ack)
{
    do
        :: sender?good_ack -> break
        :: sender?bad_ack
        :: timeout ->
            if
                :: receiver!msg;
                :: skip /* lose message */
            fi;
    od
}

inline rcv(cur_msg, cur_ack, lst_msg, lst_ack)
{
    do
        :: receiver?cur_msg -> sender!cur_ack; break /* accept */
        :: receiver?lst_msg -> sender!lst_ack
    od;
    assert(cur_ack!=ack0);
}

active procvoe Sender()

Sequence Chart: The sequence chart shows the interaction between the Sender and the Receiver. The Sender process (labeled 'Sender:0') has states 2, 6, and 4. The Receiver process (labeled 'Receiver:1') has states 3 and 4. The sequence of events is: Sender state 2 sends '1:msg1' to Receiver state 3; Receiver state 3 sends '2:ack1' to Sender state 6; Sender state 6 sends '2:ack1' to Receiver state 4.



Simulation Output: The simulation output window shows the execution trace of the simulation. The output is as follows:



```
<verification done>
<starting simulation>
spin -X -p -v -Y -g -s -r -t -j0 pan_in
spin -Z pan_in ;# preprocess input
done preprocessing

Data Values
Search for:

queue 1 ((receiver))
queue 2 ((sender))

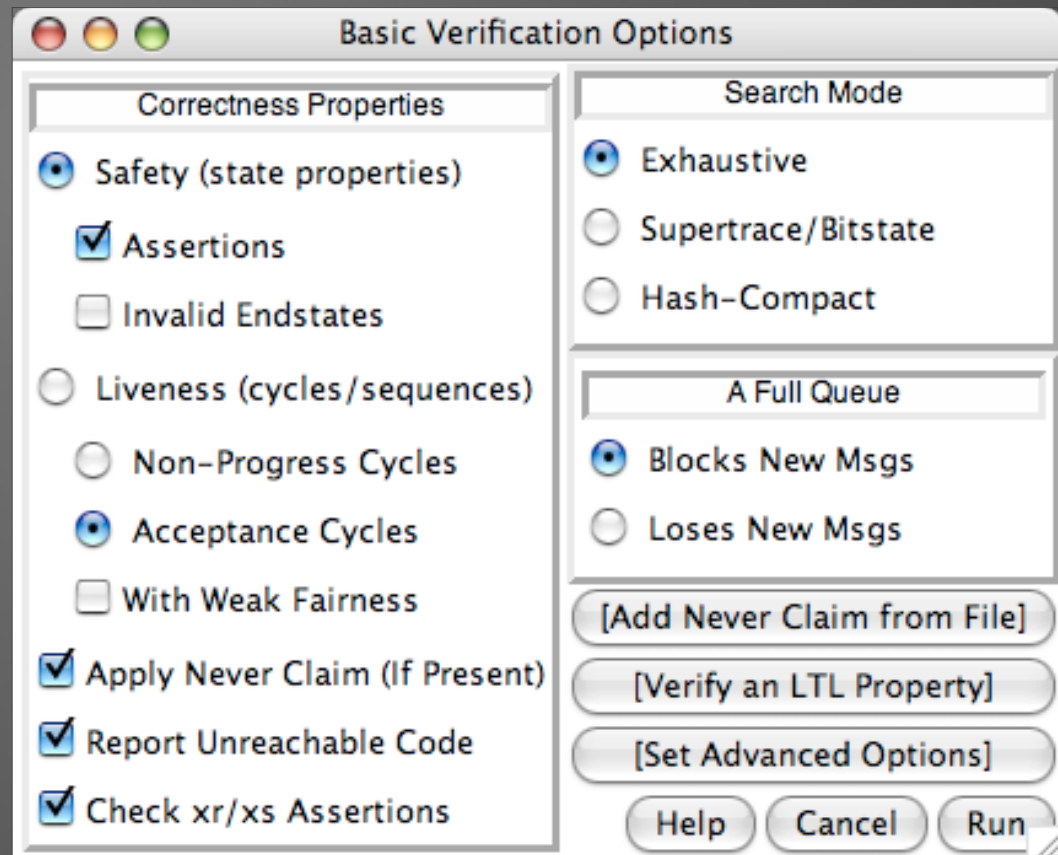
2: proc 0 (Sender) line 19 "pan_in" (state -) [values: 1!msg1]
2: proc 0 (Sender) line 19 "pan_in" (state 5) [receiver!msg1]
3: proc 1 (Receiver) line 28 "pan_in" (state -) [values: 1?msg1]
3: proc 1 (Receiver) line 28 "pan_in" (state 1) [receiver?msg1]
4: proc 1 (Receiver) line 28 "pan_in" (state -) [values: 2!ack1]
4: proc 1 (Receiver) line 28 "pan_in" (state 2) [sender!ack1]
5: proc 1 (Receiver) line 31 "pan_in" (state 9) [assert((ack1!=ack0))]
6: proc 0 (Sender) line 15 "pan_in" (state -) [values: 2?ack1]
6: proc 0 (Sender) line 15 "pan_in" (state 1) [sender?ack1]
```


```

El model checker SPIN (cont.)

Permite distintos tipos de verificaciones:

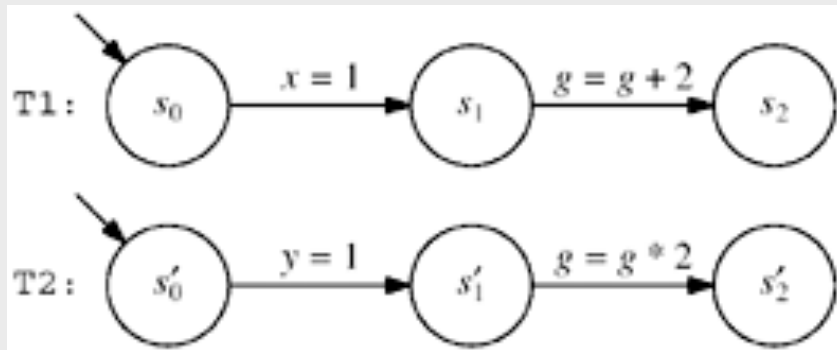
- Propiedades en LTL
- Aserciones dentro del modelo
- Deadlocks
- Progreso
- Permite verificar bajo weak fairness



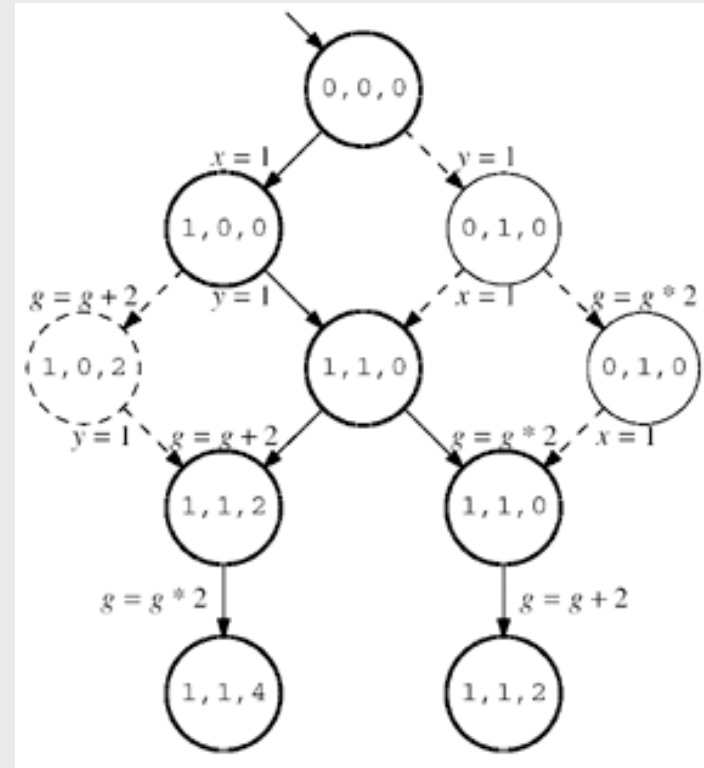
El model checker SPIN (cont.)

Técnicas de optimización

- **Bitstate hashing:** los visitados en el DFS se marcan usando una tabla hash con imagen en $\{0,1\}$.
- **Reducción por orden parcial:** aprovecha la simetría introducida por el "interleaving":



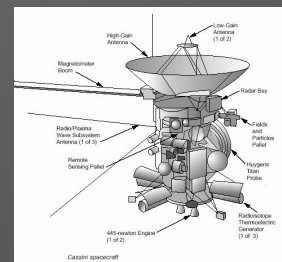
Más información y más técnicas
y algoritmos en el Cap. 9 de
"The Model Checker Spin"



El model checker SPIN (cont.)

Usos

- Spin se ha utilizado en múltiples ocasiones, y en particular, directamente en la industria (¡se implementó en la industrial!).
- Además es utilizado en la academia para aplicaciones reales (subcontratos/proyectos por parte de empresas).
- **Ejemplos:**
 - Verificación de protocolos embebidos en automotores (Bosch)
 - Verificación del dique de emergencia climática en Rotterdam.
 - Software para el procesamiento de llamadas (Lucent Tech.)
 - Diversos algoritmos en proyectos de la NASA como Deep Space 1, Cassini, Mars Exploration Rovers, Deep Impact, etc.



El model checker SMV

- SMV fue originalmente desarrollado por Ken McMillan / Edmund Clarke en Carnegie-Mellon University.
- El SMV original derivó en múltiples versiones:
 - SMV CMU (www.cs.cmu.edu/~modelcheck/smv.html)
 - SMV Cadence (*inaccessible*)
 - NuSMV (nusmv.fbk.eu) / nuXmv (nuxmv.fbk.eu)
 - => Elegir éste! (LGPL, más nuevo, único mantenido)
- Originalmente destinado a la verificación de hardware.
- Uso en línea de comandos :-)
- Manipula espacio de estados enormes.

- El lenguaje de SMV es bastante básico.
- Describe redes de autómatas (con composición sincrónica o asincrónica según se especifique).
- Descripción de cada autómata bastante declarativa, usando variables y un predicado "next" que permite hablar del valor de las variables en el siguiente estado.
- Simplemente de esa manera se definen las transiciones.

```

MODULE main
VAR
    semaphore : boolean;
    proc1 : process user(semaphore);
    proc2 : process user(semaphore);
ASSIGN
    init(semaphore) := 0;
SPEC
    AG (proc1.state = entering
        -> AF proc1.state = critical)

MODULE user(semaphore)
VAR
    state : {idle,entering,critical,exiting};
ASSIGN
    init(state) := idle;
    next(state) :=
        case
            state = idle : {idle,entering};
            state = entering & !semaphore : critical;
            state = critical : {critical,exiting};
            state = exiting : idle;
        1 : state;
    esac;
    next(semaphore) :=
        case
            state = entering : 1;
            state = exiting : 0;
        1 : semaphore;
    esac;
FAIRNESS
    running

```

```

MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;
SPEC
  AG (proc1.state = entering
    -> AF proc1.state = critical)

MODULE user(semaphore)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
    1 : state;
  esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
    1 : semaphore;
  esac;
FAIRNESS
  running

```

Simulación es
posible pero a través de la
línea de comandos

```

MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;
SPEC
  AG (proc1.state = entering
    -> AF proc1.state = critical)

MODULE user(semaphore)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
    1 : state;
  esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
    1 : semaphore;
  esac;
FAIRNESS
  running

```

Simulación es

después de la

Las propiedades se
expresan usando las lógicas
CTL, LTL, o PSL

```

MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;
SPEC
  AG (proc1.state = entering
    -> AF proc1.state = critical)

```

```

MODULE user(semaphore)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
    1 : state;
  esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
    1 : semaphore;
  esac;
FAIRNESS
  running

```

Simulación es

después de la

Las propiedades se expresan usando las lógicas CTL, LTL, o PSL

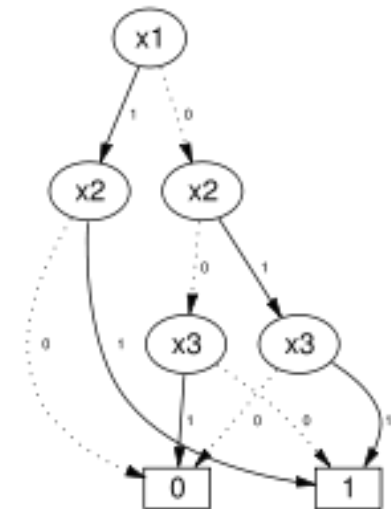
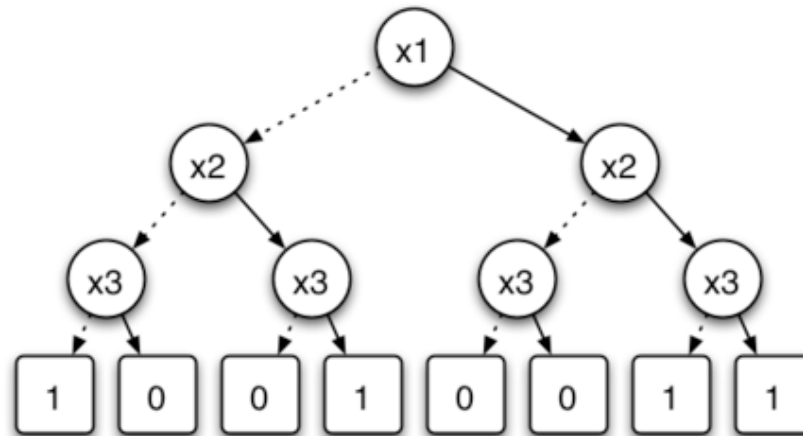
Es posible especificar que se desea hacer la verificación bajo la suposición de fairness

El model checker (Nu)SMV (cont.)

Técnicas de optimización

Representación del espacio de estado usando **BDDs**

x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



proposición a
representar
 $f \equiv (x_2 \Leftrightarrow (x_1 \vee x_3))$

Binary decision
tree

Binary decision
diagram

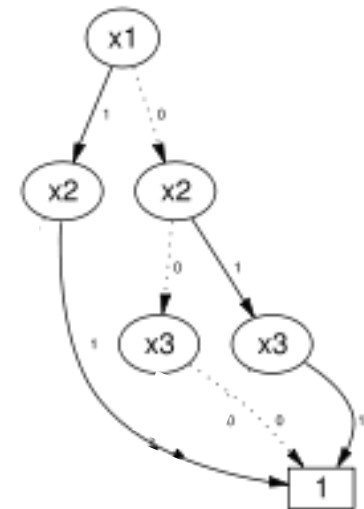
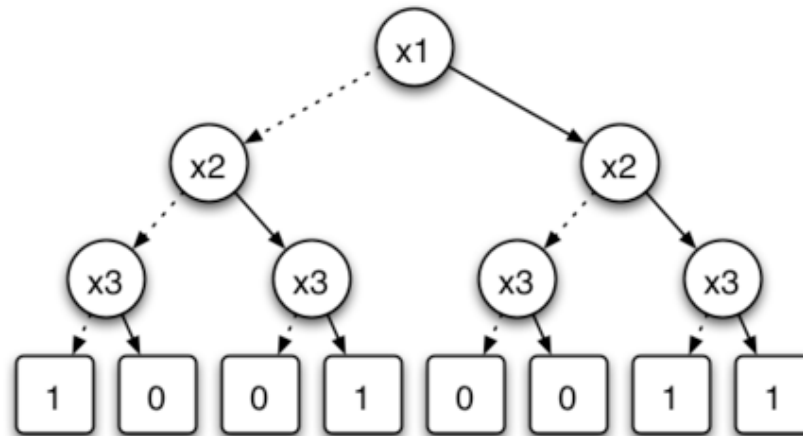
Además: **Bounded model checking** utilizando SAT solvers.

El model checker (Nu)SMV (cont.)

Técnicas de optimización

Representación del espacio de estado usando **BDDs**

x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



proposición a
representar
 $f \equiv (x_2 \Leftrightarrow (x_1 \vee x_3))$

Binary decision
tree

Binary decision
diagram

Además: **Bounded model checking** utilizando SAT solvers.

El model checker (Nu)SMV (cont.)

Usos

- Desde CMU se proveen servicios a:
 - National Science Foundation (NSF); Gigascale Systems Research Center (GSRC); Office of Naval Research (ONR); Army Research Office (ARO); Semiconductor Research Corporation (SRC); General Motors (GM).
- SMV se ha utilizado en múltiples ocasiones pero siempre desde la academia como servicio a la industria. Ej.:
 - Diversos protocolos para coherencia de cache (Gigamax, Futurebus+, etc.)
 - Diversos circuitos lógicos, componentes de procesadores y protocolos
 - Desafortunadamente no se reporta mucho en la literatura.