Métodos Formales Livianos

Marcelo Frias

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Nazareno Aguirre

Departamento de Computación

Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Universidad Nacional de Río Cuarto

Clase 5: Concurrencia y Model Checking

1 El Problema de la Corrección de Software

Poder garantizar la corrección del software que construimos es una tarea deseable. En algunas aplicaciones, es, sin duda, crucial:

- Software para equipamiento médico
- Software para el control de vehículos
- Software para el control de armamento
- (para algunos, algunas aplicaciones financieras)

⋮

A estos sistemas, cuyas fallas pueden ocasionar daños de importancia (pérdica de vidas humanas, grandes pérdidas financieras, catástrofes nucleares, etc) se denominan *sistemas críticos*.

1.1 Algunos Ejemplos

Ariane 5, 1996. Error de conversión de 64 bits a 16 bits aplicado a un valor mayor que 2^15 (pérdidas estimadas en 500 millones de dólares)

Therac-25, 1985-1987. Software de control de equipamiento médico para radioterapia. Seis personas sobre-expuestas a radiación por error en el software.

Mars Climate Orbiter, 1999. Problemas de representación (un módulo usaba sistema imperial y otro sistema decimal).

Patriot Missile, 1991. 28 muertos y 100 heridos por un error de redondeo en el software de control de Patriot Missile.

Airbus A320. Problemas en el software de control. El avión se estrella durante una demostración.

DLR's, Londres: Los trenes paraban donde no se habian construido estaciones.

Más ejemplos:

`http://www.cs.tau.ac.il/~nachumd/verify/horror.html`

2 Limitaciones de Testing y Simulación

Tanto el testing como la simulación involucran experimentos previos al lanzamiento o uso masivo del software. En general, ambos métodos proveen una serie de entradas al software, y estudian el comportamiento del mismo en esos casos.

El testing y la simulación raramente permiten garantizar la ausencia de errores. Una frase famosa al respecto: “El testing puede confirmar la presencia de errores pero nunca garantizar su ausencia”.

3 Verificación (semi)automática de Software

Existen serias limitaciones en lo que respecta a la verificación automática de software. Por ejemplo, el problema de decidir si un programa dado termina o no **no es computable**.

Sin embargo, si imponemos algunas restricciones sobre las propiedades que queremos verificar, y sobre qué sistemas, algunas tareas pueden realizarse automáticamente. Model checking es un ejemplo de esto.

4 Verificación Deductiva de Programas

La verificación de programas es la tarea de comprobar matemáticamente la corrección de un programa con respecto a su especificación. En el caso de programas secuenciales, existe desde hace varias décadas una técnica propuesta por Floyd y Hoare, basada en la especificación con aserciones en primer orden, y la visión de programas con aserciones como fórmulas de una lógica:

$$\left.\begin{array}{l} \{\texttt{Precondición}\} \\ \texttt{Programa} \\ \{\texttt{Postcondición}\} \end{array}\right\} \text{Corrección total}$$

Esta “fórmula” indica que, bajo cualquier estado que satisfaga la precondición inicialmente, la ejecución del programa termina, y lo hace en un estado que satisface la postcondición.

4.1 Ejemplos

$\{x = X \wedge y = Y\}$

```
aux := x;
x := y;
y := aux;
```

$\{x = Y \wedge y = X\}$

$\{true\}$

```
i := 1;
while (i<=length(A)) do
   j := i;
   while (j>1) do
      if (A[j-1]>A[j]) then
         swap(j-1,j)
      j := j-1
   end
   i := i+1
end
```

$\{\forall x \in [1..length(A) - 1] : A[x] \leq A[x + 1]\}$

4.2 Programas como Aserciones lógicas

Axiomas

- $\{\alpha\}$ `skip` $\{\alpha\}$
- $\{\alpha[x/X]\}$ `x := X` $\{\alpha\}$

  $\vdots$

Reglas de inferencia

- si $\{\alpha\}$ `P1` $\{\beta\}$ y $\{\beta\}$ `P2` $\{\gamma\}$ entonces $\{\alpha\}$ `P1;P2` $\{\gamma\}$
- si $\{\alpha\}$ `P` $\{\beta\}$ y $\alpha' \rightarrow \alpha$ entonces $\{\alpha'\}$ `P1` $\{\beta\}$
- si $\{\alpha\}$ `P` $\{\beta\}$ y $\beta \rightarrow \beta'$ entonces $\{\alpha\}$ `P1` $\{\beta'\}$

  $\vdots$

4.2.1 Regla de Inferencia para Iteración

La regla de inferencia para razonar sobre programas con iteración es lo que hace interesante al sistema de inferencia. Esta regla involucra nociones importantes, como las nociones de *invariante de ciclo Inv* y *función cota t*:

Si se cumplen:

(i) $\{\alpha\}$ `Init` $\{Inv\}$

(ii) $\{C \wedge Inv\}$ `CC` $\{Inv\}$

(iii) $\neg C \wedge Inv \rightarrow \beta$

(iv) $\{t = T\}$ `CC` $\{t < T\}$

(v) $C \rightarrow t > 0$

entonces

$\{\alpha\}$
```
Init;
while (C) do
  CC
end
```
$\{\beta\}$

4.3 Características de la Verificación usando Lógica de Hoare

- Puede extenderse a *programas concurrentes* (Owicki-Gries)
- **No** puede automatizarse (invariantes y funciones cotas son *mágicas*)
- Introduce conceptos de gran importancia (aserciones, especificaciones, invariantes, etc) que influyen en lenguajes y metodologías de programación

5 Programación Concurrente

Programación de sistemas compuestos de varios procesos/programas que corren concurrentemente (o en paralelo, o en forma distribuída). Muchos programas concurrentes suelen ser *reactivos*, es decir, su funcionalidad involucra la interacción permanente con el ambiente (y otros procesos).

Los sistemas reactivos tienen características diferentes a las de los programas convencionales. En muchos casos, éstos no computan resultados, y suele no requerirse que terminen (ejemplos: sistemas operativos, software de control, hardware, etc).

5.1 Interacción de Programas Concurrentes

Los programas concurrentes están compuestos por procesos (o *threads*, o componentes) que necesitan interactuar. Existen varios mecanismos de interacción entre procesos. Entre éstos se encuentran la *memoria compartida* y el *pasaje de mensajes*.

Además, los programas concurrentes deben, en general, colaborar para llegar a un objetivo común, para lo cual la *sincronización* entre procesos es crucial.

5.2 Algunos problemas comunes de programas concurrentes

- Violación de propiedades universales (invariantes)
- *Starvation* (inanición): Uno o más procesos quedan esperando indefinidamente un mensaje o la liberación de un recurso
- *deadlock*: dos o más procesos esperan mutuamente el avance del otro
- problemas de uso no exclusivo a recursos compartidos
- *livelock*: Dos o más procesos no pueden avanzar en su ejecución porque continuamente responden a los cambios en el estado de otros procesos

5.3 Concurrencia: Un Ejemplo

Consideremos el siguiente programa concurrente:

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

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

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

Qué hace este programa?

5.4 Semántica de Programas Concurrentes

Una semántica típica para programas concurrentes está basada en *sistemas de transición de estados*. Un sistema de transición de estados es un grafo en el cual:

- los nodos son los estados del sistema (posiblemente infinitos estados)
- las aristas son las transiciones atómicas de estados en estados, dadas por las sentencias del sistema.
- un subconjunto del conjunto de estados se reconoce como el conjunto de *estados iniciales*

5.4.1 Corridas de programas

Una *corrida* es una *secuencia de estados* $s_0, s_1, \ldots$, tal que:

- s_0 es un estado inicial,
- puede llegarse desde s_i s_{i+1} por alguna sentencia atómica del sistema.

Suele considerarse una transición especial denominada “stuttering” (de tartamudeo), la cual es simplemente la relación identidad entre estados. En presencia de stuttering, las ejecuciones son *siempre* infinitas, incluso si el conjunto de estados es finito.

5.4.2 Ejemplo

Volviendo a nuestro ejemplo anterior, el conjunto de estados está dado por la combinación de todos los valores posibles de las variables globales `y1`, `y2` e `in_critical`.

El estado inicial es aquel en el cual las tres variables valen 0.

Por cada sentencia atómica tenemos una transición. Por ejemplo,

```
in_critical = in_critical+1
```

relaciona todos los estados con aquellos en los cuales `y1` e `y2` no cambian su valor, e `in_critical` se incrementa en uno.

5.5 Cómo se ejecutan los procesos concurrentes?

De acuerdo al modelo computacional descripto, los procesos concurrentes se ejecutan como el *interleaving* de las acciones atómicas que los componen.

El orden en que se ejecutan las acciones atómicas no puede decidirse en general, y un mismo par de procesos pueden tener diferentes ejecuciones debido al no determinismo en la elección de las acciones atómicas a ejecutar.

6 Razonamiento Sobre Programas Concurrentes

Razonar sobre programas concurrentes es en general *muy difícil*. Luego, garantizar que un programa concurrente es correcto es también muy difícil.

Una de las razones tiene que ver con que diferentes interleavings de acciones atómicas pueden llevar a diferentes resultados o comportamientos de los sistemas concurrentes.

El número de interleavings posibles, por su parte, es en general muy grande, lo que hace que el testing difícilmente pueda brindarnos confianza de que nuestros programas concurrentes funcionan correctamente.

6.1 Abstracción: Modelos de Programas Concurrentes

Una forma de aliviar, en parte, el problema de razonar sobre programas concurrentes es considerar *representaciones abstractas* de éstos. Estas representaciones abstractas, llamadas *modelos*, nos permiten concentrarnos en las características particulares que queremos analizar.

CSP y otras álgebras de procesos permiten construir estos modelos, concentrándose en las propiedades funcionales de sistemas concurrentes. Para esto, es importante considerar los *eventos* en los cuales cada proceso puede estar involucrados, y los *patrones de ocurrencia* que éstos siguen.

6.2 El Lenguaje FSP

El lenguaje que utilizaremos en el taller para la primera parte de la asignatura es FSP (Finite State Processes). FSP es una variante simple de CSP, que incluye, entre otras cosas:

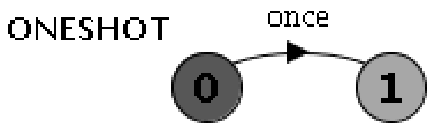
- prefijos de acciones (`x -> P`)
- Recursión (`OFF = (on ->(off -> OFF))`)
- Elección (`(x -> P | y -> Q)`)
- Elección no determinista (`(x -> P | x -> Q)`)

...

6.3 Semántica de Procesos

La semántica de los procesos FSP está dada en términos de sistemas de transición de estados y trazas, al igual que para CSP. En particular, los procesos (definidos en forma textual en CSP) pueden verse gráficamente como sistemas de transición de estados.

Por ejemplo, el proceso `ONESHOT = once -> STOP.` puede visualizarse como:



7 Tutorial de FSP y LTSA

El resto de la clase será un tutorial de FSP y su herramienta asociada LTSA (*Labelled Transition System Analyser*), haciendo énfasis en las diferencias y similitudes con CSP (un álgebra de procesos propuesta por Hoare, y una de las más difundidas).

Prefijos de Acciones. Al igual que en CSP, uno puede definir un proceso que, luego de realizar un evento o acción atómica `x`, se comporta exactamente como cierto proceso `P` usando prefijos:

```
(x -> P)
```

Esto es utilizado, por ejemplo, en el proceso `ONESHOT` visto anteriormente.

Recursión. El comportamiento de un proceso también puede definirse usando recursión, donde tenemos algunas restricciones sintácticas (similares a las de CSP):

Ejemplo:

```
SWITCH = OFF,
OFF = (on -> ON),
ON = (off -> OFF).
```

Por supuesto, estos procesos pueden verse gráficamente como sistemas de transición de estados (y la herramienta LTSA lo hace por nosotros!).

Elección. La ramificación en el flujo de corrida de un proceso se describe mediante *elección*. A diferencia de CSP, en FSP existe sólo un tipo de elección, y no puede distinguirse entre elección realizada por el sistema o por el ambiente.

Ejemplo:

```
DRINKS = ( red -> coffee -> DRINKS
         | blue -> tea -> DRINKS
         ).
```

Y, nuevamente, estos procesos pueden verse gráficamente como sistemas de transición de estados.

Elección No Determinista. Es simplemente un caso particular de elección.

Ejemplo:

```
COIN = ( toss -> heads -> COIN
       | toss -> tails -> COIN
       ).
```

Procesos y Acciones Indexados. Cuando se necesita modelar procesos que tomen un número grande de posibles valores, pueden utilizarse acciones (y procesos) indexadas, donde el rango del índice *debe* ser finito.

Esta facilidad tiene múltiples usos. En particular, puede emplearse para modelar *estado explícito* (una característica también disponible en CSP).

Ejemplos:

```
    BUFF = (in[i:0..3] -> out[i] -> BUFF).
--
    const N = 1
    range T = 0..N
    range R = 0..2*N
    SUM = (in[a:T][b:T] -> TOTAL[a+b]),
    TOTAL[s:R] = (out[s] -> SUM).
```

Acciones con Guardas. Es en general útil contar con acciones que se ejecuten condicionalmente, con respecto al estado de la máquina o sistema modelados. Esto puede expresarse usando la notación `when` en FSP.

Ejemplo:

```
COUNT (N=3) = COUNT[0],
COUNT[I:0..N] = (when(i<N) inc -> COUNT[I+1]
                |when(i>0) dec -> COUNT[I-1]
).
```

Composición Paralela. Hasta el momento, ninguna de las construcciones vistas nos permite modelar concurrencia. La construcción que nos permite hacer esto, y las más compleja de comprender, es la *composición paralela de procesos*. Ésta sigue la misma sintaxis que CSP, y, dados dos procesos `P` y `Q`, `P || Q` denota la composición paralela de estos procesos.

Ejemplo:

```
MAKER = (make -> ready -> MAKER).
USER  = (ready -> use -> USER).

||MAKER_USER = (MAKER || USER).
```

Hay varios puntos importantes por recordar con respecto a la composición paralela en FSP:

- La sincronización se realiza en las *acciones comunes*.
- A diferencia de CSP, no se puede denotar cuál es el proceso “activo” y cuál es el proceso “pasivo” en la sincronización de acciones comunes.
- El modelo de concurrencia es *interleaving*, donde las acciones atómicas independientes de diferentes procesos pueden ejecutarse en interleavings arbitrarios.

8 Recursos Compartidos

En ámbitos donde la concurrencia es útil, suele ocurrir que diferentes procesos necesitan interactuar mediante la utilización de recursos comunes.

Luego de experimentar un poco con programas concurrentes (o modelos de éstos), resulta evidente que la utilización de recursos comunes a diferentes procesos puede dar lugar a actualizaciones incorrectas en el estado de estos recursos. Este problema es conocido como *interferencia*.

Para manejar el problema, debe asegurarse que, mientras uno de los procesos utiliza un recurso compartido, el resto no puede acceder al mismo (*exclusión mutua*).

8.1 Un Ejemplo de Interferencia

Consideremos el problam del *jardín ornamental*. Tenemos un jardín ornamental con dos entradas (est y oeste), con sendas puertas giratorias. Se desea controlar el número de personas en el jardín en cada momento.

Un modelo simple del problema consiste de tres procesos: dos para controlar (independientemente) las puertas y un contador para el número de personas (recurso compartido).

```
const N = 4
range T = 0..N
set VarAlpha = {value.{read[T],write[T]}}

VAR      = VAR[0],
VAR[u:T] = (read[u]   ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
           |end   -> TURNSTILE),
INCREMENT = (value.read[x:T]
           ->value.write[x+1]->RUN)
           +VarAlpha.
```

```
||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || {east,west,display}::value:VAR)
            /{go /{east,west}.go,
              end/{east,west}.end}.
```

8.2 Detección de Errores

Para poder comprobar si el modelo anterior funciona de la manera esperada o no, podemos combinarlo con un proceso que detecte la actualización incorrecta del recurso compartido:

```
TEST        = TEST[0],
TEST[v:T]   = (when (v<N){east.arrive,west.arrive}->TEST[v+1]
              | end->CHECK[v]
              ),
CHECK[v:T] = (display.value.read[u:T] ->
                (when (u==v) right -> TEST[v]
                |when (u!=v) wrong -> ERROR)
              )+{display.VarAlpha}.
```

9 Modelando Exclusión Mutua

Una forma simple de manejar exclusión mutua es mediante una “llave”:

```
LOCK = (acquire->release->LOCK).
```

Para el caso de nuestro ejemplo, esta llave puede usarse para garantizar el acceso exclusivo al recurso compartido VAR:

```
...
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
             |end   -> TURNSTILE),
```

```
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             ->value.release->RUN
            )+VarAlpha.
...
```

10 Comunicación mediante Pasaje de Mensajes

Además de la comunicación mediante recursos compartidos, los procesos pueden comunicarse a través de *pasaje de mensajes*. Veremos un par de esquemas de pasaje de mensajes, a través de *canales*.

10.1 Pasaje de Mensajes Sincrónico

En el pasaje de mensajes a través de canales sincrónicos, la operaciones que se consideran son:

send(e,c) (se envía el dato e a través del canal c)

v = receive(c) (se recibe un dato a través del canal c y se lo almacena en v)

Pasaje de Mensajes Sincrónico (cont.)

Cuando un proceso necesita ejecutar una operación de comunicación a través de un canal sincrónico (ya sea de envío o recepción), éste se bloquea hasta tanto otro proceso realice la acción complementaria.

Luego de la comunicación, los procesos continúan independientemente.

Para modelar las operaciones de envío y recepción de mensajes a través de canales sincrónicos en FSP podemos usar acciones indexadas:

send(e,chan) se modela con `chan[e]`

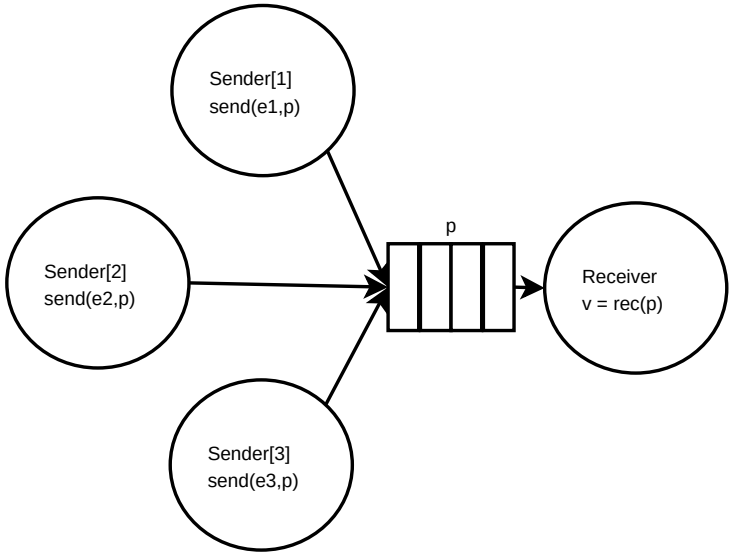
v = receive(chan) se modela con `chan[e: M]`

10.2 Ejemplo:

```
range M = 0..9

SENDER = SENDER[0],
SENDER[e:M] = (chan.send[e]->SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]->RECEIVER).

||SyncMsg = (SENDER || RECEIVER)/{chan/chan.{send,receive}}.
```

10.3 Pasaje de Mensajes Asincrónico

En el pasaje de mensajes asincrónico, la acción de envío no se bloquea, y los mensajes son almacenados en buffers, hasta que las acciones de recepción son ejecutadas.



Pasaje de Mensajes Asincrónico (cont.)

Para modelar las operaciones de envío y recepción de mensajes a través de canales asincrónicos en FSP debemos considerar buffers acotados (debe preservarse la finitud de los modelos). Para hacer esto puede utilizarse un modelo de *puertos*:

```
range M = 0..9
set   S = {[M],[M][M]}


PORT            //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]       //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
```

```
PORT[t:S][h:M]  //two or more  messages queued to port
   = (send[x:M]->PORT[x][t][h]
     |receive[h]->PORT[t]
     ).
```

10.4 Ejemplo:

```
range M = 0..9
set S = {[M],[M][M]}
PORT             = (send[x:M]->PORT[x]),
PORT[v:M]        = (send[x:M]->PORT[x][v]
                   |receive[v]->PORT),
PORT[s:S][v:M]   = (send[x:M]->PORT[x][s][v]
                   |receive[v]->PORT[s]).

ASENDER = ASENDER[0],
ASENDER[e:M] = (port.send[e]->ASENDER[(e+1)%10]).
```

```
ARECEIVER = (port.receive[v:M]->ARECEIVER).

||AsyncMsg = (s[1..2]:ASENDER || ARECEIVER||port:PORT)
             /{s[1..2].port.send/port.send}.

||Abstract = AsyncMsg
             /{s[1..2].port.send/s[1..2].port.send[M],
               port.receive/port.receive[M]
              }.
```