Métodos Formales Industriales (MFI)

—prácticas—

Grado de Ingeniería en Informática

Práctica 3: Model checking en NuSMV utilizando lógica CTL

Santiago Escobar

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 NuSMV.

El sistema NuSMV es una herramienta de validación de sistemas de estados finitos respecto a especificaciones en la lógica temporal CTL. El lenguaje de entrada de NuSMV se ha diseñado para permitir descripciones de sistemas de estados finitos tanto síncronas como asíncronas, es decir, máquinas (de Mealy) síncronas o redes abstractas asíncronas de procesos indeterministas. El lenguaje, además, permite realizar descripciones modulares de sistemas a la vez que definiciones de componentes reutilizables. Los únicos tipos de datos disponibles son finitos, ya que permiten describir máquinas de estados finitos. La lógica CTL facilita la definición de propiedades temporales, como seguridad, vivacidad, equidad y ausencia de bloqueos. NuSMV implementa el algoritmo de model checking simbólico basado en el uso de OBDDs, que determina de forma eficiente si el programa satisface una propiedad expresada en la lógica CTL.

Primero vamos a proporcionar una pequeña introducción sobre la lógica temporal CTL utilizada para luego proceder a explicar el sistema NuSMV.

2. Computation Tree Logic (CTL)

Computation tree logic (CTL) es un 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 CTL debe llevar primero un operador de estado, seguido de un operador de caminos y, por último, una proposición lógica.

- Operadores de estado. Tomando un estado concreto del sistema como estado actual, determinamos la satisfacibilidad de una proposición lógica p con respecto a los caminos que surgen del estado actual.

- Fórmula "$A\ p$" (All): la proposición p debe satisfacerse en todos los caminos que surjan del estado actual.
- Fórmula "$E\ p$" (Exists): la proposición p debe satisfacerse en al menos uno de los caminos que surjan del estado actual.

- Operadores de caminos. Fijando un estado actual y un camino a partir de él, determinamos la satisfacibilidad de una proposición 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 CTL:

1. $AG\ p$. Siempre llueve.
2. $AF\ p$. Siempre llegará un momento en que llueva.
3. $EF\ p$. Algunas veces llueve.
4. $EG\ p$. Puede ocurrir que no pare nunca de llover.
5. $AX\ p$. Va a llover en el siguiente instante de tiempo, pase lo que pase.
6. $EX\ p$. Puede que empiece a llover en el siguiente instante de tiempo.
7. $A\ (q\ U\ p)$. Siempre juego al tenis hasta que empieza a llover.
8. $E\ (q\ U\ p)$. A veces juego al tenis hasta que empieza a llover.
9. $EF\ (AG\ p)$. Puede que empiece a llover y en tal caso no parará nunca, pase lo que pase.
10. $AG\ (EF\ p)$. Siempre ocurre que algunas veces llueve.
11. $AF\ (EG\ p)$. Siempre es posible que empiece a llover y no pare nunca.
12. $EG\ (AF\ p)$. Es posible que llegue un momento en el que lloverá ocurra lo que ocurra.

3. El sistema NuSMV

El sistema NuSMV comprueba que un sistema de estados finitos satisface la especificación asociada proporcionada en la lógica CTL. El lenguaje incluido en NuSMV permite describir la relación de transición de una estructura Kripke finita. A continuación se describen algunas de las cualidades del sistema NuSMV:

- *Módulos.* El usuario puede descomponer la descripción de un sistema complejo de estados finitos en módulos. Cada módulo puede instanciarse de múltiples formas y sus variables pueden ser accedidas desde otros módulos, es decir, no hay ocultación de información. Los módulos pueden tener parámetros, que pueden ser variables de un estado, expresiones u otros módulos. Además, los módulos pueden incluir restricciones de equidad como fórmulas de la lógica CTL (lo cual no es posible en otros lenguajes de validación).

- *Composición síncrona o asíncrona.* Se pueden realizar composiciones síncronas o asíncronas de módulos en NuSMV. En una composición síncrona, cada paso del sistema corresponde a un paso en cada uno de los componentes o módulos, mientras que en la asíncrona, un paso en la composición representa un paso en uno y sólo uno de los componentes. Si se especifica un módulo con la palabra clave `process`, se utiliza el modo asíncrono para ese módulo, es decir, se ejecuta como un proceso paralelo al proceso `main`; mientras que en caso contrario se sobrentiende el modo síncrono.

- *Transiciones indeterministas.* Las transiciones de estados en un modelo pueden ser deterministas o indeterministas. El indeterminismo refleja una elección entre las posibles acciones modeladas en el sistema. También se puede utilizar para describir una versión más abstracta de un sistema donde determinados detalles son irrelevantes.

- *Relación de transición.* La relación de transición de un módulo se puede especificar bien como relaciones de términos que definen explícitamente el valor actual y siguiente de las variables de estado, o implícitamente como un conjunto de asignaciones de valor ejecutadas paralelamente. Las sentencias de asignación paralelas definen el valor de las variables en el siguiente estado en términos de los valores que poseen en el estado actual.

El sistema SMV fue desarrollado desde 1992 por la Universidad Carnegie Mellon, situada en Pittsburgh, PA (USA), y estaba accesible en la URL:

`http://www-2.cs.cmu.edu/~modelcheck/SMV.html`

Desde 2001, el sistema SMV ha sido reimplementado y extendido con nuevas funcionalidades como "linear time logic" o "bounded model checking", pasando a llamarse NuSMV, y está accesible en la URL:

`http://NuSMV.fbk.eu`

En esta práctica, nos centraremos en la sintaxis original de SMV, aunque utilizaremos NuSMV en el laboratorio.

El sistema NuSMV está disponible para Windows y diferentes versiones de UNIX (incluyendo Linux y Mac OS X). El programa no necesita ninguna instalación previa y su ejecución se realiza llamándolo directamente desde el shell:

```
$ nusmv prueba
```

donde, de esta forma, acepta un sistema concurrente descrito en NuSMV y realiza la validación de las propiedades incluidas en la especificación. Si la propiedad a verificar es correcta, NuSMV devolverá simplemente true. Si la propiedad es falsa, NuSMV devolverá un contraejemplo que demuestra que dicha propiedad es incorrecta para la especificación.

3.1. Sintaxis de NuSMV

Un programa en NuSMV está compuesto de diferentes módulos. Cada módulo es una descripción encapsulada que puede incluir parámetros formales y que puede instanciarse varias veces. Por ejemplo, la declaración:

```
MODULE user(semaphore)
```

define `user` como un módulo con 1 parámetro formal.

Se pueden declarar variables locales dentro de un módulo. Los tipos de variables permitidos son **finitos**: booleanos, tipos enumerados, subrangos dentro de los enteros o vectores de elementos.

```
VAR state: {idle, entering, critical, exiting};
    position: 0..2;
    elements: array 0..1 of boolean;
```

Además, las variables pueden ser instancias de módulos, de esta forma un módulo puede contener instancias de otros módulos y crear una estructura jerárquica entre ellos. Por ejemplo, el módulo `main` declara la variable `proc1` como una instancia de `user` y además la define como un proceso asíncrono que se ejecuta en paralelo al módulo `main`:

```
MODULE main
VAR proc1: process user(semaphore);
```

Cada especificación (programa) en NuSMV debe contener un módulo llamado `main` sin parámetros. Este módulo define el módulo principal en la jerarquía y el punto de inicio para la construcción del modelo de estados finitos asociado a la especificación.

El valor de las variables de estado se define usando las palabras clave `init` y `next` dentro de la sección `ASSIGN` del módulo:

```
ASSIGN
   init(state) := idle;
   next(state) := case
      (state = idle): {idle, entering};
      (state = entering) & !semaphore: critical;
   ...
   esac;
```

El código anterior define el valor inicial de la variable `state` como `idle` y el valor en el estado siguiente como condiciones expresadas en términos de los valores de las variables en el estado actual. Si el valor actual de la variable `state` es `idle`, la posibilidad de hacer uso de la elección indeterminista nos permite expresar que el valor de dicha variable en el siguiente estado será `idle` o `entering`. Si, en cambio, el valor actual de `state` es `entering` y el valor actual de la variable `semaphore` es falso, el valor de la variable `state` en el siguiente estado será `critical`. De esta forma estamos haciendo uso de sentencias de asignación que se ejecutan en paralelo para obtener el valor de las variables en el siguiente estado del sistema.

El valor de una variable en el siguiente estado puede definirse incluso en función del valor de otra variable en ese siguiente estado. Es decir, el siguiente ejemplo:

```
ASSIGN
   next(state) := case
```

```
   ...
      (state = waiting) & (next(tray) = closed): preparing;
   ...
   esac;
```

indica que el valor de la variable `state` en el siguiente estado es `preparing` si en el estado actual está esperando a cerrar el cajón de papel y en el siguiente estado el cajón va a estar ya cerrado. **Cuidado**, porque no se permiten definiciones circulares de variables, como en el siguiente ejemplo:

```
ASSIGN
   next(x) := case
      (next(y) != y): FALSE;
   esac;
   next(y) := case
      (next(x) != x): TRUE;
   esac;
```

Las propiedades que deben satisfacerse se especifican como fórmulas de la lógica CTL y van asociadas a la palabra clave `SPEC`. Por ejemplo, si los dos procesos `proc1` y `proc2` no pueden estar a la vez en su zona crítica, determinaremos esta propiedad dentro del módulo `main` de la siguiente forma:

```
SPEC AG !(proc1.state = critical & proc2.state = critical)
```

Se pueden incluir restricciones de equidad ("fairness"), que permiten reducir el número de caminos considerados por NuSMV, como una fórmula de la lógica CTL utilizando la palabra clave `FAIRNESS`. Cada proceso posee una variable booleana `running` que indica cuándo el proceso se está ejecutando y que puede utilizarse para definir restricciones de equidad. Por ejemplo, el módulo `proc` indica que sólo deben considerarse aquellos caminos en los cuales el proceso llegue a ejecutarse un número indeterminado de veces.

```
MODULE proc
...
FAIRNESS running
```

Aparte de esta breve presentación de NuSMV, podéis consultar una descripción completa y detallada del sistema en el manual de usuario disponible en

`http://NuSMV.fbk.eu`

También se dispone de un tutorial, ejemplos, etc.

4. Ejemplo de especificación NuSMV

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:

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := FALSE;
SPEC -- safety
  AG !(proc1.state = critical & proc2.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 : exiting;
      state = exiting : idle;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering & !semaphore : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;
FAIRNESS
  running
```

Si llevamos a cabo su verificación con NuSMV obtenemos el siguiente resultado:

```
$ nusmv eje1
-- specification AG (!(proc1.state = critical & proc2.sta... is true
```

Este resultado muestra que el problema de la mutua exclusión de dos procesos es resuelto por la especificación, es decir, los dos procesos no pueden estar en su parte crítica al mismo tiempo.

Sin embargo, si incluimos las siguientes propiedades en el ejemplo anterior, que describen que todo intento de un proceso de acceder a su zona crítica tiene éxito:

```
SPEC -- liveness (proc1)
  AG (proc1.state = entering -> AF proc1.state = critical)
SPEC -- liveness (proc2)
  AG (proc2.state = entering -> AF proc2.state = critical)
```

obtenemos que dichas propiedades no son satisfechas por la especificación. Esto queda expresado por cada uno de los respectivos contraejemplos que entrega NuSMV, donde básicamente uno de los procesos siempre entra antes que el otro en su zona crítica:

```
-- specification AG (proc1.state = entering -> AF proc1.state = critical)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
```

```
Trace Type: Counterexample
-> State: 1.1 <-
  semaphore = FALSE
  proc1.state = idle
  proc2.state = idle
-> Input: 1.2 <-
  _process_selector_ = proc1
  running = FALSE
  proc2.running = FALSE
  proc1.running = TRUE
-- Loop starts here
-> State: 1.2 <-
  proc1.state = entering
-> Input: 1.3 <-
  _process_selector_ = proc2
  running = FALSE
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.3 <-
  proc2.state = entering
-> Input: 1.4 <-
  _process_selector_ = proc2
  running = FALSE
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.4 <-
  semaphore = TRUE
  proc2.state = critical
-> Input: 1.5 <-
  _process_selector_ = proc1
  running = FALSE
  proc2.running = FALSE
  proc1.running = TRUE
-> State: 1.5 <-
-> Input: 1.6 <-
  _process_selector_ = proc2
  running = FALSE
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.6 <-
  proc2.state = exiting
-> Input: 1.7 <-
  _process_selector_ = proc2
  running = FALSE
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.7 <-
  semaphore = FALSE
  proc2.state = idle
```

Se aconseja intentar entender la traza de contraejemplo devuelta por SMV, prestando atención a los estados "`State: 1.X`"donde se muestra qué variables han cambiado del estado anterior a éste y las marcas de "`Loop starts here`"que indican una transición del último estado de la traza de vuelta al estado inmediatamente posterior a la marca.

5. Objetivo de la práctica

El objetivo de esta práctica consiste en responder a la serie de preguntas cortas que se muestran a continuación, modificando los programas de la forma requerida. Vamos a utilizar el proyecto QUIZIFY de PSW como inspiración para los ejercicios de toda la asignatura.

Evaluación: La evaluación de esta práctica se realizará subiendo las respuestas a la tarea de PoliformaT.

5.1. Lista de preguntas de un quiz

Los módulos de Maude de las prácticas anteriores modelan una lista de preguntas de un quiz o encuesta de QUIZIFY. Esa lista tendrá tantas posiciones como preguntas tenga el quiz. El símbolo ? se añadió a la lista para indicar que es una pregunta aún sin respuesta, el símbolo * se añadió para indicar que era una pregunta que quería pasar a modo respuesta y el símbolo # se añadió para indicar que era una pregunta en modo respuesta. Sin embargo, SMV no acepta esos símbolos y tenemos que utilizar constantes más sencillas: `noanswer`, `request` y `answering`. Hemos asumido solamente cuatro respuestas en los quiz o encuestas: A, B, C y D.

Al igual que en la práctica anterior, debemos asegurarnos que solo uno de los clientes está visitando la propiedad a la vez. Para ello podemos adaptar el ejemplo de QUIZIFY de la práctica anterior de Maude a la sintaxis de SMV usando el ejemplo guía de los dos procesos con un semáforo de la sección anterior.

```
MODULE main
VAR
  semaphore : boolean;
  question1 : process question(semaphore);
  question2 : process question(semaphore);
ASSIGN
  init(semaphore) := FALSE;
SPEC
  AG !(question1.state = answering & question2.state = answering) ---safety
SPEC
  AG (question1.state = request -> AF question1.state = answering) ---liveness1
SPEC
  AG (question2.state = request -> AF question2.state = answering) ---liveness2

MODULE question(semaphore)
VAR
  state : {noanswer,request,answering,A,B,C,D};
ASSIGN
  init(state) := noanswer;
  next(state) :=
    case
      state = noanswer : {noanswer,request};
      state = request & !semaphore : answering;
      state = answering : {A,B,C,D};
      state = A : {A,request};
      state = B : {B,request};
      state = C : {C,request};
      state = D : {D,request};
      TRUE : state;
    esac;
```

```
  next(semaphore) :=
    case
      state = request & !semaphore : TRUE;
      state = answering : FALSE;
      TRUE : semaphore;
    esac;
FAIRNESS
  running
```

Hemos decidido complicar un poco el modelo con ciclos infinitos permitiendo que una pregunta respondida pueda pasar a `request`.

Al verificar este programa, nos pasa exactamente lo mismo que el ejemplo de los dos procesos con un semáforo de la sección anterior. Es decir, la propiedad de safety se satisface pero las dos propiedades de vivacidad no se satisfacen.

(Pregunta 1) Modifica el código anterior con una variable turno, p.ej. "`turn : 1..2`", y añade un identificador de pregunta del quiz, '`MODULE question(id,semaphore,turn)`", para resolver el problema de la vivacidad. Fíjate que tendrás que añadir un `next(turno)` en el módulo `question` o en el módulo `main`.

(Pregunta 2) ¿Cómo has modificado las condiciones de `next(state)` para que tenga encuenta el turno y el identificador? Razona tu respuesta.

(Pregunta 3) ¿Cómo has creado `next(turno)` para que tenga encuenta el turno, el estado de una pregunta y su identificador? Existen varias alternativas. Razona tu respuesta.

(Pregunta 4) ¿Se te ha ocurrido usar que una pregunta pueda saber qué va a hacer en el siguiente estado? Razona tu respuesta.

```
next(turn) :=
  case
     ...
    turn = 1 & state = noanswer & next(state) = noanswer : 2;
    ...
  esac;
```

(Pregunta 5) ¿Cómo de difícil es extender el código para más de dos preguntas? Razona tu respuesta.

5.2. Modelado de lista de preguntas con un protocolo

Se complica la situación y nos dicen que en la realidad no es tan sencillo, porque hay un mensaje que cada dispositivo envía a un servidor central y éste debe responder dando permiso. Se nos ha ocurrido escribir el siguiente trozo de código en SMV. Al igual que en las prácticas anteriores, los dispositivos no se modelan pero sí sus interacciones con el servidor. El canal de comunicaciones de las prácticas anteriores ahora se representa de forma explícita con un proceso nuevo. El servidor también se representa de forma explícita, como cada una de las preguntas.

Por ejemplo, se nos ha ocurrido definir un proceso `channel` que representa un canal de comunicaciones entre una pregunta y el servidor.

```
MODULE channel()
VAR
  q2s : {empty,request,finished}; --- question to server
  s2q : {empty,granted}; --- server to question
ASSIGN
  init(q2s) := empty; --- Initialization
  init(s2q) := empty; --- Initialization
  next(q2s) := q2s; --- Keep current value
  next(s2q) := s2q; --- Keep current value
FAIRNESS
  running
```

Este proceso lleva fijados los mensajes que cada pregunta y el servidor pueden generar y recibir a través del canal. La lógica interna del canal es sencilla, simplemente repite de forma indefinida el contenido de las variables `q2s` y `s2q`, question-to-server y server-to-question, respectivamente. La parte más interesante es cómo enlazar las dos preguntas y el servidor con los dos canales.

```
MODULE main
VAR
  ch1 : process channel(); --- Channel question 1 to/from server
  ch2 : process channel(); --- Channel question 2 to/from server
  server : process server(turn,ch1.q2s,ch1.s2q,ch2.q2s,ch2.s2q);
  question1 : process question(ch1.s2q,ch1.q2s);
  question2 : process question(ch2.s2q,ch2.q2s);
  turn : 1..2;
```

Para asegurar la propiedad de exclusión mutua, el servidor mantiene información de quién está en modo respuesta y espera el mensaje `finished` por parte de esa pregunta. Además, ahora es el servidor quien se encarga del turno con una variable `chosen` para asegurar las propiedades de vivacidad.

```
MODULE server(turn,input1,output1,input2,output2)
VAR
  chosen : 0..2;
ASSIGN
  init(chosen) := 0; --- No question chosen
  next(chosen) :=
    case
      input1 = request & input2 != request & chosen = 0 : 1;
      input1 != request & input2 = request & chosen = 0 : 2;
      input1 = request & input2 = request & turn = 1 & chosen = 0 : 1;
      input1 = request & input2 = request & turn = 2 & chosen = 0 : 2;
      input1 = finished | input2 = finished : 0;
      TRUE : chosen;
    esac;
  next(turn) :=
    case
      chosen = 1 : 2;
      chosen = 2 : 1;
      TRUE : turn;
    esac;
  next(input1) :=
    case
      input1 = request & next(chosen) = 1 : empty;
      input1 = finished & next(chosen) = 0 : empty;
```

```
      TRUE : input1;
    esac;
  next(input1) :=
    case
      input2 = request & next(chosen) = 2 : empty;
      input2 = finished & next(chosen) = 0 : empty;
      TRUE : input2;
    esac;
  next(output1) :=
    case
     ....
    esac;
  next(output2) :=
    case
     ....
    esac;
FAIRNESS
  running
```

El módulo `question` ya no necesita ni el identificador ni el turno, aunque siga implementando la misma lógica interna de antes.

```
MODULE question(input,output)
VAR
  state : {noanswer,request,answering,A,B,C,D};
ASSIGN
  init(state) := noanswer;
  next(state) :=
    case
      state = noanswer : {noanswer,request};
      state = request & input = granted : answering;
      state = answering : {A,B,C,D};
      state = A : {A,request};
      state = B : {B,request};
      state = C : {C,request};
      state = D : {D,request};
      TRUE : state;
    esac;
  next(input) :=
    case
      input = granted & next(state) = answering : empty;
      TRUE : input;
    esac;
  next(output) :=
    case
     ....
    esac;
  ....
FAIRNESS
  running
```

(Pregunta 6) Completa el código anterior para que se sigan satisfaciendo la propiedad de safety y las dos propiedades de vivacidad.

(Pregunta 7) ¿Has sincronizado de forma correcta los cambios de la variable `state` con los mensajes enviados a través del parámetro `output`? Razona tu respuesta.

(Pregunta 8) ¿Has sincronizado de forma correcta los cambios de la variable `chosen` con los mensajes enviados a través del parámetro `output`? Razona tu respuesta.

(Pregunta 9) ¿Te has asegurado de que cada pregunta y el servidor consuman los mensajes que les llegan? Razona tu respuesta.

5.3. Modelado de lista de preguntas con pérdida de mensajes

Se complica aún más la situación y nos dicen que el canal de comunicación es muy inestable y algunos mensajes se pierden y la persona de la empresa pide un modelo más tolerante a fallos. La idea de que los mensajes se pierdan la expresamos añadiendo un valor de `error` e indicando que el canal puede perder el valor de `q2s` o `s2q`.

```
MODULE channel()
VAR
  q2s : {error,empty,request,finished}; --- question to server
  s2q : {error,empty,granted}; --- server to question
ASSIGN
  init(q2s) := empty; --- Initialization
  init(s2q) := empty; --- Initialization
  next(q2s) := {error,q2s}; --- Error or keep current value
  next(s2q) := {error,s2q}; --- Error or keep current value
FAIRNESS
  running
```

(Pregunta 10) Ahora las propiedades de vivacidad no se cumplen.

```
AG (question1.state = request -> AF question1.state = answering)
AG (question2.state = request -> AF question2.state = answering)
```

Es decir las propiedades descritas como "*Para todo estado donde la pregunta* 1 *(resp.* 2*) esté en modo* ***request****, cualquier camino que surja del estado satisface que eventualmente la pregunta* 1 *(resp.* 2*) esté en modo* ***answering***".

Pero, en vez de modificar el programa para satisfacerlas, reescribe esas dos fórmulas CTL para que representen la siguiente definición: "*Para todo estado donde la pregunta* 1 *(resp.* 2*) esté en modo* ***request****, cualquier camino que surja del estado satisface que o bien se pierde el contenido del canal en algún momento (valor* ***error****) o el contenido del canal no se pierde y, en ese caso, la pregunta* 1 *(resp.* 2*) eventualmente llega al modo* ***answering***". La idea es utilizar una expresión "until".