Manual de introducción a NuSMV

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 SMV.

El sistema SMV es una herramienta de validación de sistemas de estados finitos respecto a especificaciones en la lógica temporal CTL. El lenguaje de entrada de SMV 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. SMV 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 utilizada por SMV para luego proceder a explicar el sistema SMV.

2. Lógica de tiempo ramificado

La *lógica de tiempo ramificado* (CTL por sus siglas en inglés) es un lógica temporal utilizada para expresar propiedades termporales 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 satisfactibilidad 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 satisfactibilidad de una proposición 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 en el camino fijado.
- Fórmula $\Diamond\ p$ (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. $A \Box p$ indica que siempre llueve.
2. $A \Diamond p$ indica que siempre llegará un momento en que llueva.
3. $E \Diamond p$ indica que algunas veces llueve.
4. $E \Box p$ indica que puede ocurrir que no pare nunca de llover.
5. $A \bigcirc p$ indica que va a llover en el siguiente instante de tiempo, pase lo que pase.
6. $E \bigcirc p$ indica que puede que empiece a llover en el siguiente instante de tiempo.
7. $A (q \ U \ p)$ indica que siempre juego al tenis hasta que empieza a llover.
8. $E (q \ U \ p)$ indica que a veces juego al tenis hasta que empieza a llover.
9. $E \Diamond (A \Box p)$ indica que puede que empiece a llover y en tal caso no parará nunca, pase lo que pase.
10. $A \Box (E \Diamond p)$ indica que siempre ocurre que algunas veces llueve.
11. $A \Diamond (E \Box p)$ indica que siempre es posible que empiece a llover y no pare nunca.
12. $E \Box (A \Diamond p)$ indica que es posible que llegue un momento en el que lloverá ocurra lo que ocurra.

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

1. $\Diamond \Box p$ indica que siempre llegará un momento en que llueva y no pare de llover. Esto se puede describir con LTL pero no con CTL.
2. $A \Box (E \Diamond p)$ indica que para todo instante de tiempo, existe la posibilidad de que llueva. Esto se puede describir con CTL pero no con LTL.

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

3. El sistema NuSMV

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

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 SMV. 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 solo 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 ha sido desarrollado desde 1992 por la Universidad Carnegie Mellon, situada en Pittsburgh, PA (USA), y está 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 este documento, nos centraremos en la sintaxis 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.smv
```

donde, de esta forma, acepta un sistema concurrente descrito en SMV y realiza la validación de las propiedades incluídas 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 SMV

Un programa en SMV 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 SMV 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 `false`, 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 SMV, 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 solo 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 SMV, 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 en SMV

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 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.smv
-- 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 incluímos 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
Este problema puede solucionarse incluyendo:
FAIRNESS
    state = critical
```

en el proceso `user`, aparte de la restricción de ejecución. Sin embargo, en ese caso estamos restringiendo el comportamiento del sistema y, si no se corresponde con el comportamiento a modelar, estaríamos definiendo una especificación incorrecta.