Generación semiautomática de cursos de Electrónica para Internet mediante un lenguaje de simulación continua orientado a objetos

Alfonseca, Manuel; de Lara, Juan; Pulido, Estrella
Departamento de Ingeniería Informática, Universidad Autónoma de Madrid
Ctra. de Colmenar, km.15, 28049-MADRID
Tel.-91-3974409 Fax.- 91-3975277
{Manuel.Alfonseca, Juan.Lara, Estrella.Pulido}@ii.uam.es

RESUMEN

Este artículo describe OOCSMP, un lenguaje de simulación especialmente aplicable cuando el modelo puede descomponerse en varias componentes similares en interacción mutua. OOCSMP, una extensión orientada a objetos del lenguaje CSMP, también ha sido extendido para resolver ecuaciones en derivadas parciales. Los compiladores desarrollados generan código C++ o Java que puede visualizarse en diversos sistemas operativos. Además se han desarrollado procedimientos que permiten generar cursos para Internet basados en la simulación de sistemas.

1. Introducción

La simulación de sistemas [1] es una herramienta estándar en los programas multimedia y en la enseñanza asistida por ordenador. Hemos desarrollado procedimientos y herramientas que nos permiten generar, fácil y semiautomáticamente, cursos para la Universidad basados en la simulación de sistemas. Estos cursos están formados por páginas HTML y applets Java y se pueden invocar a través de Internet.

2. El lenguaje OOCSMP

El lenguaje que usamos para la programación de los modelos (OOCSMP), ha sido diseñado por nosotros como una extensión orientada a objetos de CSMP [2], y se ha descrito en artículos anteriores [3-5]. Algunas de sus características lo hacen muy útil para la simulación de sistemas que constan de muchas componentes similares que interactúan entre sí.

El lenguaje OOCSMP permite utilizar directamente los siguientes bloques básicos:

- Generadores de ondas: escalón, rampa, impulso, onda cuadrada, diente de sierra, sinusoide, ruido aleatorio uniforme, ruido gaussiano.
- Modificadores de ondas: limitador, zona muerta, cuantificador, bloque con histéresis.
- Generadores de funciones matemáticas: funciones trigonométricas e hiperbólicas y sus inversas, exponencial, logaritmos decimal y neperiano, raíz cuadrada, residuo, valor absoluto, y el máximo o mínimo de varias entradas.
- Generadores de funciones arbitrarias: por interpolación lineal o cuadrática, pendiente de una función arbitraria.
- Bloques lógicos: puertas AND, OR, OR exclusivo, NAND, NOR, NOT (inversor), biestables RST y JK.
- Interruptores: interruptor de entrada, interruptor funcional, mantenedor (hold) de orden cero.
- Bloques varios: integrador, derivador, retardo, función implícita, comparador, equivalencia.

OOCSMP dispone de otras posibilidades, como la selección de métodos de integración adecuados, elegidos entre varios disponibles (rectangular, trapezoidal, Simpson, Adams, Runge-Kutta), la resolución de ecuaciones en derivadas parciales de una familia muy amplia (por el método de diferencias finitas o el de elementos finitos), y la elección de diversos formatos de salida:

- Listados
- Gráficos lineales, animados o no
- Gráficos tridimensionales
- Gráficos icónicos de diversos tipos

3. Modelos utilizados en el curso sobre Electrónica

El proyecto que describimos en este artículo es un curso práctico sobre Electrónica, en el que se describe la construcción de módulos complejos partiendo de otros más simples. También es posible combinar circuitos lógicos con componentes eléctricos ordinarios, cosa que no suele estar contemplada por los simuladores corrientes.

Utilizando los bloques básicos de proceso lógico de OOCSMP, detallados más arriba, construimos primero un sumador de un bit, como indica el listado 1.

```
*************************************************
* clase ADDER1.CSM : implementa un sumador de 1 bit
*************************************************
class ADDER1 {
*************************************************
* entradas : A y B los dos bits a sumar, C el acarreo
* salidas : OUT y CARRY
*************************************************
  DYNAMIC A,B,C
    xor1 := EOR ( A,B )
    and1 := AND ( A,B )
    and2 := AND ( and1, C )
    CARRY:= OR ( and1, and2 )
    OUT  := EOR ( xor1, C )
}
```

Listado 1: Definición de un sumador de un bit en función de los bloques básicos

El sumador de un bit (ADDER1) se ha definido como una clase, utilizando las facilidades de orientación a objetos del lenguaje OOCSMP. Para todos los efectos prácticos, ADDER1 es ahora un nuevo bloque de OOCSMP, utilizable de la misma manera que los bloques básicos incluidos en el lenguaje. Tal como se ha definido, es un bloque de tres entradas (las variables A, B y C, declaradas en la instrucción DYNAMIC) y dos salidas (las variables CARRY y OUT). En nuestra versión actual de OOCSMP, no es necesario declarar las salidas, pues cualquiera de las variables internas a los objetos de una clase es accesible desde el exterior del bloque, y por tanto puede utilizarse como salida. En versiones posteriores se podrían introducir especificadores de acceso, para que algunas de las variables de un objeto puedan hacerse inaccesibles desde fuera.

El listado 2 muestra cómo se construye un sumador de cuatro bits utilizando cuatro módulos sumadores de un bit conectados en serie.

```
*****************************************************
* clase ADDER4.CSM : implementa un sumador de 4 bits
*****************************************************
INCLUDE "ADDER1.CSM"
class ADDER4
{
    ADDER1 a1()
    ADDER1 a2()
    ADDER1 a3()
    ADDER1 a4()
  DATA
    OUT[4]
*****************************************************
* entradas : A y B son dos vectores de longitud 4,
* que son los dos nos. a sumar, C es el acarreo
*****************************************************
  DYNAMIC A[], B[], C
    a1.STEP ( A[0] , B[0] , C )
    a2.STEP ( A[1] , B[1] , a1.CARRY )
    a3.STEP ( A[2] , B[2] , a2.CARRY )
    a4.STEP ( A[3] , B[3] , a3.CARRY )
    CARRY := a3.CARRY
    OUT[0] := a1.OUT
    OUT[1] := a2.OUT
    OUT[2] := a3.OUT
    OUT[3] := a4.OUT
*****************************************************
* salidas : OUT y CARRY
*****************************************************
}
```

Listado 2: Definición de un sumador de cuatro bits en función de cuatro sumadores de un bit

Obsérvese que los objetos de una clase pueden tener como componentes objetos de otra clase. El comportamiento de un objeto *x* se simula mediante la instrucción *x.STEP(valores de entrada)*.

Veamos otro ejemplo: el listado 3 presenta la definición de un decodificador 2x4.

```
*****************************************************
* Esta clase implementa un decodificador de 2x4
*****************************************************
CLASS DEC2x4
{
*****************************************************
* Las salidas se guardan en un vector de 4 componentes.
```

```
*******************************************************
DATA D[4]
*******************************************************
* Las entradas del decodificador : X0,X1,E
* ( E habilita el decodificador )
*******************************************************
DYNAMIC X0,X1,E
noX0  := NOT ( X0 )
noX1  := NOT ( X1 )
D[0]    := AND ( E, noX0, noX1 )
D[1]    := AND ( E, noX0, X1 )
D[2]    := AND ( E, X0, noX1 )
D[3]    := AND ( E, X0, X1 )
*******************************************************
* La salida del decodificador es D[]
*******************************************************
}
```

Listado 3: Definición de un decodificador 2x4 en función de los bloques básicos

El listado 4 muestra la utilización de varios decodificadores 2x4 para construir un decodificador 4x16.

```
*******************************************************
* Esta clase implementa un decodificador de 4x16, utilizando
* 5 decodificadores 2x4 en forma de árbol
*******************************************************
INCLUDE "DEC2x4.CSM"
CLASS DEC4x16
{
*******************************************************
* Los decodificadores que vamos a usar
*******************************************************
DEC2x4 root ()
DEC2x4 leaf1 ()
DEC2x4 leaf2 ()
DEC2x4 leaf3 ()
DEC2x4 leaf4 ()
*******************************************************
* El vector donde vamos a guardar la salida
*******************************************************
DATA Z[16]
*******************************************************
* Las entradas del decodificador : Y[],E (E habilita el
* decodificador , y Y es un array de 4 componentes)
*******************************************************
DYNAMIC Y[],ON
root.STEP (Y[3],Y[2],ON)
leaf1.STEP ( Y[1],Y[0],root.D[0])
leaf2.STEP ( Y[1],Y[0],root.D[1])
leaf3.STEP ( Y[1],Y[0],root.D[2])
leaf4.STEP ( Y[1],Y[0],root.D[3])
*******************************************************
* Las salidas del decodificador : Z0...Z15
*******************************************************
Z[0]:=leaf1.D[0]
Z[1]:=leaf1.D[1]
Z[2]:=leaf1.D[2]
Z[3]:=leaf1.D[3]
Z[4]:=leaf2.D[0]
Z[5]:=leaf2.D[1]
Z[6]:=leaf2.D[2]
Z[7]:=leaf2.D[3]
Z[8]:=leaf3.D[0]
Z[9]:=leaf3.D[1]
Z[10]:=leaf3.D[2]
Z[11]:=leaf3.D[3]
Z[12]:=leaf4.D[0]
Z[13]:=leaf4.D[1]
```

```
Z[14]:=leaf4.D[2]
Z[15]:=leaf4.D[3]
}
```

Listado 4: Definición de un decodificador 4x16 en función de decodificadores 2x4

4. Generación semiautomática de cursos para la Internet

Hemos desarrollado un compilador que traduce los modelos escritos en OOCSMP a código C++ o Java y páginas HTML. Estas últimas deben completarse manualmente con el texto, imágenes e iconos apropiados. Dependiendo de las opciones del compilador, podemos hacer que éste:

- Genere código C++ para probar el modelo bajo DOS.
- Genere código C++ para probar el modelo en Windows-95, Unix o MacIntosh, usando las librerías Amulet, desarrolladas por la Universidad de Carnegie Mellon [6].
- Genere código Java para probar el modelo en un intérprete de Java o en un visor de "applets".
- Genere código Java e incluya el "applet" en una página HTML.
- Añada automáticamente botones para seleccionar ejecuciones alternativas del modelo.
- Genere ventanas y "widgets" especiales para modificar los valores de los parámetros del modelo y hacer posible experimentos del tipo "¿qué pasaría si ...?".

Los pasos que seguimos para la construcción del curso son los siguientes:

1. Diseñar el curso sobre el papel.
2. Diseñar la simulación a realizar en cada página.
3. Programar los modelos en OOCSMP.
4. Generar automáticamente código en C++ y probar los modelos.
5. Generar automáticamente "applets" Java y esqueletos HTML.
6. Añadir manualmente textos, imágenes y referencias complementarias a las páginas HTML.

5. El curso de Electrónica

En cada página del curso, el alumno puede observar (en una tabla o de forma gráfica) la evolución de la salida de los módulos ante las diversas entradas que se le presenten, así como experimentar con los modelos.

Las páginas disponibles actualmente muestran ejemplos progresivamente más complejos, como la construcción de un sumador de cuatro bits a partir de sumadores de un bit, o la de un decodificador 4x16 a partir de decodificadores 2x4 (conectados en forma de árbol). Otros módulos disponibles son: multiplexores, codificadores y biestables. El curso incluye además una página de experimentación libre.

El formato gráfico que representa de forma esquemática los circuitos sobre la pantalla se genera automáticamente a partir de las ecuaciones. Este circuito no es pasivo, pues el usuario puede interaccionar con él, cambiando los valores de las entradas y comprobando interactivamente los efectos de dichos cambios sobre la salida y las puertas lógicas

intermedias. El curso puede encontrarse en la siguiente dirección de red: http://www.ii.uam.es/~epulido/circ/modules.htm.

6. Conclusión

Las herramientas y métodos propuestos permiten :

- Desarrollar fácilmente cursos educativos para Internet.
- Facilitar la modificación de los modelos y la prueba de los mismos en distintas situaciones.
- El profesor puede desarrollar variaciones en el funcionamiento del modelo, generándose un botón para la ejecución de cada una de ellas.
- Los modelos construidos no sólo pueden utilizarse como parte del curso, sino como programas C++ con otros objetivos.
- Es posible combinar circuitos lógicos y analógicos.

En el futuro tenemos intención de permitir que el alumno pueda construir y probar circuitos cada vez más complejos mediante un editor gráfico interactivo escrito en Java, que extienda las capacidades actuales de la representación gráfica de circuitos que se utiliza en las páginas de nuestro curso.

7. Bibliografía

[1] Y. Monsef, "Modelling and Simulation of Complex Systems", SCS Int., Erlangen, 1997.

[2] IBM Corp.: "Continuous System Modelling Program III (CSMP III) and Graphic Feature (CSMP III Graphic Feature) General Information Manual", IBM Canada, Ontario, GH19-7000, 1972.

[3] M. Alfonseca, E. Pulido, J. de Lara, R. Orosco, "OOCSMP: An Object-Oriented Simulation Language", *Proc. 9th European Simulation Symposium ESS'97*, pp. 44-48, 1997.

[4] M. Alfonseca, J. de Lara, E. Pulido, "Semiautomatic Generation of Educational Courses in the Internet by Means of an Object-Oriented Continuous Simulation Language", *Proc. 12th European Simulation Multiconference ESM'98*, pp. 547-551, 1998.

[5] M. Alfonseca, R. Carro, J. de Lara, E. Pulido, "Education in Ecology at the Internet with an Object-Oriented Simulation Language", *Proc. Eurosim'98 Simulation Congress*, Fed. European Simulation Societies, ed. K.Juslin, pp. 118-123, 1998.

[6] B. A. Myers, E. Borison, A. Ferrency, R. McDaniel, R. C. Miller, A. Faulring, B. D. Kyle, P. Doane, A. Mickish, A. Klimovitski, "The Amulet V3.0 Reference Manual", Carnegie Mellon University School of Computer Science Technical Report no. CMU-CS-95-166-R2 and Human Computer Interaction Institute Technical Report CMU-HCII-95-102-R2, Marzo 1997.