

Trabajo Práctico 2

Generación dinámica de fractales

Grupo 1

Julio Kriger
L.U.: 207/69
jkriger@dc.uba.ar

4 de agosto de 2002

<i>ÍNDICE</i>	1
---------------	---

Índice

1. Introducción	2
2. Primera parte - Autómata Celular	3
3. El modelo conceptual	3
3.1. Modelos atómicos	3
3.1.1. GeneradorParticulas	3
3.1.2. GeneradorRandom	4
3.1.3. BRW	4
3.2. Modelo acoplado	4
4. Especificación de los modelos atómicos	5
4.1. GeneradorParticulas	5
4.2. GeneradorRandom	5
4.3. BRW	6
5. Especificación del modelo acoplado	11
6. Resultados	12
7. Segunda parte - Modelo DEVS	13
8. El modelo conceptual	13
8.1. Modelos atómicos	13
8.1.1. GeneradorParticulas	13
8.1.2. BRW	13
8.2. Modelo acoplado	13
9. Especificación de los modelos atómicos	14
9.1. GeneradorParticulas	14
9.2. BRW	15
10. Especificación del modelo acoplado	21
11. Resultados	22
12. Conclusión	23
Índice de figuras	24
Referencias	24

1. Introducción

El presente trabajo consta de dos partes. La primera parte explica la generación dinámica de fractales a través de un autómata celular implementado en *DEVS*, y la segunda parte explica la generación a través de un Modelo DEVS. Esta segunda parte fue desarrollada luego de la finalización de la primera parte. Se revió en profundidad dicha parte y se logro un nuevo modelo, una evolución dentro del marco DEVS. Ambas partes son posibles soluciones al problema planteado, aunque una es mas eficiente que la otra.

2. Primera parte - Autómata Celular

A continuación se expondrá la primera parte del trabajo, la generación dinámica de fractales a través de un Autómata Celular implementado en DEVS.

3. El modelo conceptual

El sistema elegido a elaborar en el presente trabajo es una sistema de generación dinámica de fractales. Originalmente el sistema estaba basado en el modelo *Diffusion Limited Aggregation*[1][2] o DLA, sin embargo para simplificarlo se lo transformo a un modelo sobre *Branching Random Walk*[3][4] o BRW.

La idea de BRW y DLA es usar el método de Montecarlo para la generación del fractal. Dada una matriz de $(n \times n)$ se tiran m partículas al azar por techo (primera fila), luego esa partícula se mueve en una trayectoria al azar hacia arriba, abajo, izquierda y derecha, rebotando contra el techo y las paredes laterales (primer y última columna). La partícula se *pega* si choca contra otra partícula ya pegada o si llega al piso (la última fila). La generación de números al azar se realiza mediante un generador de números *uniformes* para los rangos $[0, \text{cantidad de columnas de la matriz})$ y $[0, 1)$, el primer rango es de números enteros y el segundo es de números reales.

Al contrario que en los modelos DLA, en los modelos BRW los caminos al azar están sesgados por una probabilidad de arrastre DP (*drift probability*), que modifica la estructura del fractal. DP es un real comprendido en el rango $[0, 1)$.

Se puede resumir el movimiento de la partícula de la siguiente manera, dado un número al azar P real $[0, 1)$, una probabilidad de arrastre DP , y la posición de la partícula (i, j) :

1. Si $0 \leq P < 0,25 * (1 - DP)$, mover a $(i - 1, j)$
2. Si $0,25 * (1 - DP) \leq P < 0,5 * (1 - DP)$, mover a $(i, j + 1)$
3. Si $0,5 * (1 - DP) \leq P < 0,75 * (1 - DP)$, mover a $(i, j - 1)$
4. Si $0,75 * (1 - DP) \leq P < 1$, mover a $(i + 1, j)$

3.1. Modelos atómicos

A continuación se explicarán los modelos atómicos observados.

3.1.1. GeneradorParticulas

Este modelo genera números enteros uniformes al azar entre 0 y cantidad de columnas menos uno, que se utilizarán para ubicar nuevas partículas en la matriz. Es un modelo muy simple al cual hay que indicarle, mediante dos parámetros, la cantidad de partículas a generar y la cantidad de columnas de la matriz. Al recibir un evento de pedido de nueva partícula controla que aún queden partículas a generar, de ser así genera una nueva partícula sino no genera nuevas partículas.

3.1.2. GeneradorRandom

Este modelo genera números reales uniformes al azar entre $[0,1)$ que se utilizarán para mover la partícula por la matriz. En este simple modelo a cada evento de pedido de nueva número al azar genera un número al azar.

3.1.3. BRW

Éste es el modelo *CELL-DEVS* que contiene a la matriz, recibe eventos de número al azar y nueva partícula, genera eventos de nuevo número al azar y nueva partícula, contiene toda la lógica de pegado y movimiento de partículas.

3.2. Modelo acoplado

Existe un único modelo acoplado, que acopla a los tres modelos atómicos antes mencionados. La figura 1 muestra un esquema del modelo acoplado. El modelo acepta un solo evento de entrada: *start*, que da comienzo a la simulación. Este modelo coordina la necesidad del modelo BRW de nuevas partículas y números al azar con el generador de partículas y números al azar.

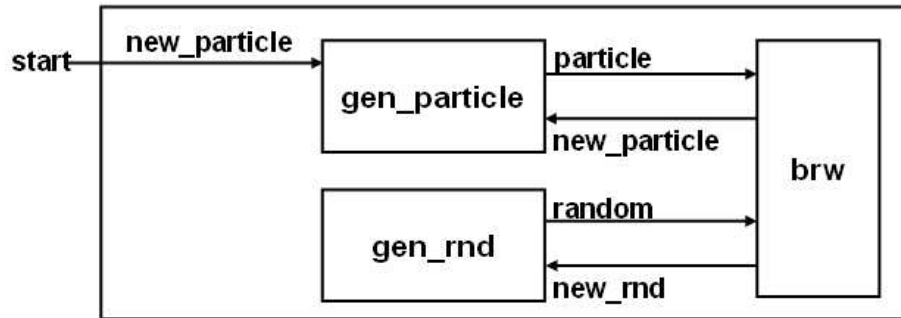


Figura 1: BRW

4. Especificación de los modelos atómicos

A continuación se especificarán los modelos atómicos descritos anteriormente. Si bien no se realizará una especificación formal como la dada en clase, se utilizará una especificación alternativa para una fácil y comprensible lectura.

Primeramente se declaran los *eventos entrantes y salientes*. Luego, los *estados* se declaran como nombre de variable y entre paréntesis sus posibles valores. Por último, sigue un pseudocódigo de las *funciones de transición externa, interna y de salida*.

4.1. GeneradorParticulas

Eventos Entrantes

new_particle

Eventos Salientes

particle

Estados

columns (1...n), particulas (1...m)

Función de transición externa

```
Si llega evento new_particle
  Si particulas > 0
    decrementar particulas en 1
    pasar a estado Activo, ta(Activo) = 20 ms
    (significa la programacion de un evento
     interno para dentro de 20 ms)
```

Función de transición interna

Pasivar

Función de Salida

```
Generar numero entero al azar entre 0 y (columns - 1)
y enviar ese numero generado con el evento particle.
```

4.2. GeneradorRandom

Eventos Entrantes

new_rnd

Eventos Salientes

random

Estados

Función de transición externa

```
Si llega evento new_rnd
  pasar a estado Activo, ta(Activo) = 50 ms
  (significa la programacion de un evento
   interno para dentro de 50 ms)
```

Función de transición interna

Pasivar

Función de Salida

Generar numero real al azar entre $[0, 1)$ y enviar ese numero generado con el evento random.

4.3. BRW

A continuación se especificará el modelo atómico CELL-DEVS descrito anteriormente. Si bien no se realizará una especificación formal como la dada en clase, se utilizará una especificación alternativa para una fácil y comprensible lectura.

Este modelo es del tipo celular con una demora de transporte reflejando una matriz plana, sus bordes no están unidos como en una figura *toroidal*.

El ejemplo que se expondrá será de una matriz cuadra de (16×16) . No obstante, se pueden llevar sin ninguna complicación a una matriz cuadra de $(n \times n)$.

Eventos Entrantes

particle, random

Eventos Salientes

new_particle, new_rnd

Vecindad

La vecindad puede ser mejor entendida observando la figura 2. El porque de esta vecindad se explicará al reglas de movimiento descritas más abajo.

	(-2,-1)	(-2,0)	(-2,1)	
(-1,-2)	(-1,-1)	(-1,0)	(-1,1)	(-1,2)
(0,-2)	(0,-1)	(0,0)	(0,1)	(0,2)
(1,-2)	(1,-1)	(1,0)	(1,1)	(1,2)
	(2,-1)	(2,0)	(2,1)	

Figura 2: Vecindad de la celda $(0,0)$ **Estados de las celdas**

Las celdas pueden tener como estado valores 0, 1 o 2. Inicialmente la matriz toda esta inicializa en estado 0.

- El valor 0 indica que es una celda vacía (sin partícula alguna).
- El valor 1 indica que es una celda con una partícula pegada.
- El valor 2 indica que es una celda con una partícula en movimiento

Distribución de los eventos de entrada a las celda

Al llegar el evento *particle* al modelo BRW, éste lo distribuye a todas las celda de la primer fila, llamando a las reglas de transición *new_particle_position*. Ver figura 3.

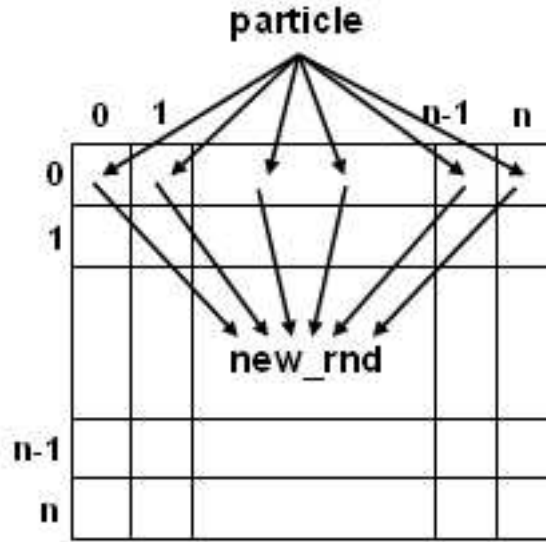


Figura 3: Distribución del evento *particle*

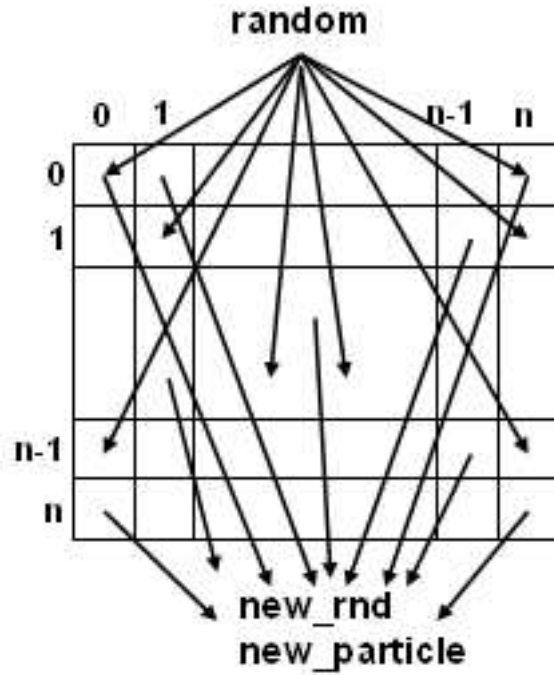
Las siguientes reglas pertenecientes a la transición *new_particle_position* solo hacen que la celda cuya posición (columna) sea igual al número azar del evento pase a tener valor 2 y disparar el evento de generar un valor al azar para mover la nueva partícula. Si la celda no tiene el mismo valor del número al azar del evento, no se modifica y queda con su valor anterior.

[*new_particle_position*]

```
rule : { 2 + send(new_rnd,-999) } 0 { (portValue(thisPort) =
CELLPOS(1)) } rule : { (0,0) } 0 { t }
```

Al llegar el evento *random* al modelo BRW, éste lo distribuye a todas las celda de la matriz, llamando a las reglas de transición *new_random*. Ver figura 4.

Las siguientes reglas pertenecientes a la transición *new_random* hacen mover a la partícula hacia arriba, abajo, izquierda o derecha según corresponda con el número al azar del evento, o le cambian de valor a 0 o 1 dependiendo de los valores de sus vecinos.

Figura 4: Distribución del evento *random*

[new_random]

```
rule : { 1 + send(new_particle,-111) } 0 { ((0,0) = 2) AND
((CELLPOS(0) = 7) OR ( ((-1,-1) = 1) OR ((1,-1) = 1) OR ((-1,1) =
1) OR ((1,1) = 1) OR ((1,0) = 1) OR ((0,1) = 1) OR ((-1,0) = 1) OR
((0,-1) = 1) ) ) }
```

Esta regla cambia el valor de una celda de 2 a 1, es decir, la *pega* si es una celda con valor 2 y está en la última fila o tiene algún vecino inmediato en una vecindad de Moore con valor 1.

```
rule : { 2 + send(new_rnd,-222) } 0 { ((0,0) = 2) AND (CELLPOS(0)
= 0) AND (0 <= portValue(thisPort)) AND (portValue(thisPort) <
0.125) }
```

```
rule : { 2 + send(new_rnd,-333) } 0 { ((0,0) = 2) AND (CELLPOS(1)
= 7) AND (0.125 <= portValue(thisPort)) AND (portValue(thisPort) <
0.25) }
```

```
rule : { 2 + send(new_rnd,-444) } 0 { ((0,0) = 2) AND (CELLPOS(1)
= 0) AND (0.25 <= portValue(thisPort)) AND (portValue(thisPort) <
0.375) }
```

Estas tres reglas hacen que si la partícula choca contra el techo o una pared lateral, se que en el lugar. Por ejemplo si la celda tiene valor 2, esta en el techo, y el azar dice que debe moverse la partícula hacia arriba, la partícula rebota y queda en su lugar. Lo mismo para las paredes laterales (si la celda tiene valor 2 y esta en una pared lateral izquierda o derecha y el azar dice que debe moverse la partícula a izquierda o derecha respectivamente la partícula rebota y queda en su lugar).

Las siguientes reglas dicen como se mueve la partícula. A diferencia de las reglas anteriores, la celda a la cual la partícula se debe mover debe tener valor 0 y controlar que realmente esa celda se puede mover libremente y no que se pegue en el próximo paso de tiempo.

```
rule : { 2 + send(new_rnd,-555) } 0 { ((0,0) = 0) AND ((1,0) = 2)
AND (0 <= portValue(thisPort)) AND (portValue(thisPort) < 0.125)
AND ((0,1) != 1) AND ((1,1) != 1) AND ((2,1) != 1) AND ((2,0) !=
1) AND ((2,-1) != 1) AND ((1,-1) != 1) AND ((0,-1) != 1) AND
(CELLPOS(0) != 6) }
```

Esta regla mueve la partícula una celda hacia arriba. Si la celda tiene valor 0, la celda de abajo tiene valor 1, el azar dice que debe moverse la partícula hacia arriba, no estoy en la ante-última fila (es decir, la partícula esta el piso por lo que se paga en próximo paso de tiempo, así que no hay que mover dicha partícula) y la celda de abajo (la que tiene la partícula) no tiene ningún vecino con valor 1 en una vecindad de Moore (es decir, en el próximo paso de tiempo no pasará a valer 1) entonces se cambia el valor de la celda a 2 y se dispara el evento de nuevo numero al azar.

```
rule : { 2 + send(new_rnd,-666) } 0 { ((0,0) = 0) AND ((0,-1) = 2)
AND (0.125 <= portValue(thisPort)) AND (portValue(thisPort) <
0.25) AND ((1,0) != 1) AND ((1,-1) != 1) AND ((1,-2) != 1) AND
((0,-2) != 1) AND ((-1,-2) != 1) AND ((-1,-1) != 1) AND ((-1,0) !=
1) AND (CELLPOS(0) != 7) }
```

Esta regla mueve la partícula una celda hacia la derecha. Si la celda tiene valor 0, la celda a izquierda tiene valor 1, el azar dice que debe moverse la partícula hacia la derecha, no estoy en la última fila (es decir, la partícula esta el piso por lo que se paga en próximo paso de tiempo, así que no hay que mover dicha partícula) y la celda a izquierda (la que tiene la partícula) no tiene ningún vecino con valor 1 en una vecindad de Moore (es decir, en el próximo paso de tiempo no pasará a valer 1) entonces se cambia el valor de la celda a 2 y se dispara el evento de nuevo numero al azar.

```
rule : { 2 + send(new_rnd,-777) } 0 { ((0,0) = 0) AND ((0,1) = 2)
AND (0.25 <= portValue(thisPort)) AND (portValue(thisPort) <
0.375) AND ((-1,0) != 1) AND ((-1,1) != 1) AND ((-1,2) != 1) AND
((0,2) != 1) AND ((1,2) != 1) AND ((1,1) != 1) AND ((1,0) != 1)
AND (CELLPOS(0) != 7) }
```

Esta regla mueve la partícula una celda hacia la izquierda. Si la celda tiene valor 0, la celda a derecha tiene valor 1, el azar dice que debe moverse la partícula hacia la izquierda, no estoy en la última fila (es decir, la partícula esta el piso por lo que se paga en próximo paso de tiempo, así que no hay que mover dicha partícula) y la celda a derecha (la que tiene la partícula) no tiene ningún vecino con valor 1 en una vecindad de Moore (es decir, en el próximo paso de tiempo no pasará a valer 1) entonces se cambia el valor de la celda a 2 y se dispara el evento de nuevo numero al azar.

```
rule : { 2 + send(new_rnd,-888) } 0 { ((0,0) = 0) AND ((-1,0) = 2)
AND (0.375 <= portValue(thisPort)) AND (portValue(thisPort) < 1)
AND ((0,-1) != 1) AND ((-1,-1) != 1) AND ((-2,-1) != 1) AND
((-2,0) != 1) AND ((-2,1) != 1) AND ((-1,1) != 1) AND ((0,1) != 1)
}
```

Esta regla mueve la partícula una celda hacia abajo. Si la celda tiene valor 0, la celda de arriba tiene valor 1, el azar dice que debe moverse la partícula hacia abajo y la celda de arriba (la que tiene la partícula) no tiene ningún vecino con valor 1 en una vecindad de Moore (es decir, en el próximo paso de tiempo no pasará a valer 1) entonces se cambia el valor de la celda a 2 y se dispara el evento de nuevo numero al azar.

```
rule : { 0 } 0 { ((0,0) = 2) }
```

Esta regla *limpia* la celda con valor 2 y la deja en 0. Hay que tener en cuenta que si se llega a valer esta regla es porque no hay ninguna celda con valor 1 en una vecindad de Moore y no se esta en el piso.

```
rule : { (0,0) } 0 { t }
```

Esta regla deja las celdas como estaban originalmente, esta regla se ejecuta siempre que una celda tenga valor 1 o no hayan valido ninguna de las reglas anterior.

Tie-breaking

No debería ocurrir nunca una superposición de eventos, sin embargo se definió que el evento *new_particle* tenga prioridad sobre el evento *new_rnd*.

5. Especificación del modelo acoplado

A continuación se especificará el modelo acoplado descrito anteriormente. Si bien no se realizará una especificación formal como la dada en clase, se utilizará una especificación alternativa para una fácil y comprensible lectura.

Primeramente se declaran los *eventos entrantes y salientes*. Luego, los *componentes* se declaran como nombre del componente y entre paréntesis su tipo. Por último, se declara la forma en que los componentes se acoplan, *acoplamiento de eventos entrantes, salientes e internos*. Ver figura 1.

Eventos Entrantes

`start`

Eventos Salientes

Componentes

```
gen_particle (GeneradorParticulas),
gen_rnd (GeneradorRandom),
brw (brw)
```

Acoplamiento de eventos entrantes

```
start --> gen_particle.new_particle
```

Acoplamiento interno

```
gen_particle.particle --> brw.particle
gen_rnd.random --> brw.random
brw.random brw.new_particle --> gen_particle.new_particle
brw.new_rnd --> gen_particle.new_rnd
```

Acoplamiento de eventos salientes

Tie-breaking

El orden de prioridad de los eventos es es siguiente:

1. *start*
2. *particle*
3. *random*
4. *new_particle*
5. *new_rnd*

6. Resultados

Se realizaron varias pruebas sobre matrices de diferentes tamaños (16×16), (32×32), (64×64) y (128×128) y con varias probabilidades de arrastre ($DP = \{0.0, 0.3, 0.5 \text{ y } 0.8\}$). Todos los scripts para realizar las pruebas y los resultados se encuentran en el disco adjunto.

El modelo se comporto según lo especificado. Al trazar x caminos al azar, los tiempos de las simulaciones de las pruebas no se comportaron de manera homogénea. Sin embargo, cabe destacar que las simulaciones corridas con mayor DP tardaron menos tiempo en finalizar que las simulaciones con DP menor. Esto es una clara consecuencia de la formula utilizada.

Dado al tiempo requerido para la simulación, solo se realizaron las pruebas para los modelos con matrices de (16×16) con las cuatro probabilidades de arrastre. Al comparar las matrices resultante se observo un *achataamiento* a medida que la probabilidad de arrastre. Esto es consistente con el modelo. No se adjuntan los archivos de log resultante de la simulación ya que son muy extensos, sí se adjunta en un archivo zip los archivos “.drw” resultantes de la aplicación *drawlog*.

7. Segunda parte - Modelo DEVS

A continuación se expondrá la segunda parte del trabajo, la generación dinámica de fractales a través de un Modelo DEVS.

8. El modelo conceptual

Esta nueva versión del modelo BRW presenta una mejora en la performance del tiempo real de ejecución de la simulación, una gran disminución en el número de mensajes utilizados y un archivo de log de menor tamaño.

Se eliminó el modelo atómico *GeneradorRandom* y se reemplazo por la función intrínseca *random* de C++.

La idea de ésta nueva versión es realizar dos acciones en una:

1. Mover la partícula de una celda a otra.
2. Generar un número al azar para mover nuevamente la partícula.

8.1. Modelos atómicos

A continuación se explicarán los modelos atómicos observados que han sido modificados para esta nueva versión.

8.1.1. GeneradorParticulas

Este modelo genera números enteros uniformes al azar entre 0 y cantidad de columnas menos uno, que se utilizarán para ubicar nuevas partículas en la matriz. Es un modelo muy simple al cual hay que indicarle, mediante dos parámetros, la cantidad de partículas a generar y la cantidad de columnas de la matriz. Al recibir un evento de pedido de nueva partícula controla que aún queden partículas a generar, de ser así genera una nueva partícula sino no genera nuevas partículas.

8.1.2. BRW

Este modelo CELL-DEVS contiene la matriz, recibe un evento de partícula, genera un evento de nueva partícula, contiene toda la lógica de pegado y movimiento de partículas.

8.2. Modelo acoplado

Existe un único modelo acoplado, que acopla a los modelos atómicos *GeneradorParticula* y *BRW*. La figura 5 muestra un esquema del nuevo modelo acoplado. El modelo acepta un solo evento de entrada: *start*, que da comienzo a la simulación. Este modelo coordina la necesidad del modelo BRW de nuevas partículas con el generador de partículas.

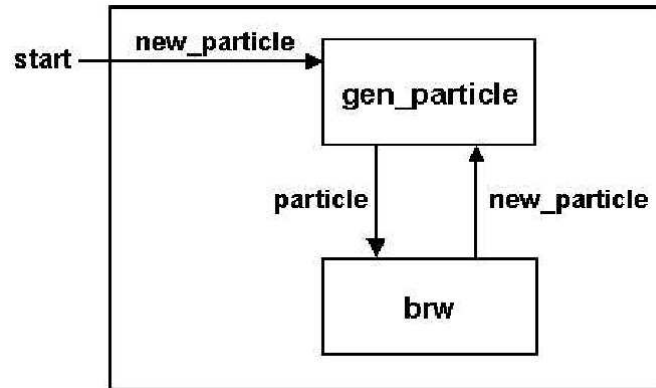


Figura 5: Nuevo modelo BRW

9. Especificación de los modelos atómicos

A continuación se especificarán los modelos atómicos descritos anteriormente. Si bien no se realizará una especificación formal como la dada en clase, se utilizará una especificación alternativa para una fácil y comprensible lectura.

Primeramente se declaran los *eventos entrantes y salientes*. Luego, los *estados* se declaran como nombre de variable y entre paréntesis sus posibles valores. Por último, sigue un pseudocódigo de las *funciones de transición externa, interna y de salida*.

9.1. GeneradorParticulas

Eventos Entrantes

new_particle

Eventos Salientes

particle

Estados

columnas (1...n), particulas (1...m)

Función de transición externa

```

Si llega evento new_particle
  Si particulas > 0
    decrementar particulas en 1
    pasar a estado Activo, ta(Activo) = 20 ms
    (significa la programacion de un evento
     interno para dentro de 20 ms)
  
```

Función de transición interna

Pasivar

Función de Salida

Generar numero entero al azar entre 0 y (columnas - 1) y enviar ese numero generado con el evento *particle*.

9.2. BRW

A continuación se especificará el nuevo modelo atómico CELL-DEVS. Si bien no se realizará una especificación formal como la dada en clase, se utilizará una especificación alternativa para una fácil y comprensible lectura.

Este modelo es del tipo celular con una demora de transporte reflejando una matriz plana, sus bordes no están unidos como en una figura toroidal.

El ejemplo que se expondrá será de una matriz cuadrada de (16×16) . No obstante, se pueden llevar sin ninguna complicación a una matriz cuadrada de $(n \times n)$.

Dado que el modelo se comporta como un autómata celular, todas las reglas tienen el tiempo de programación de un evento interno en $20ms$, inclusive el modelo de generador de partículas.

Los valores de la formula de movimiento, por ejemplo $0,25 * (1 - DP)$, se encuentran pre-calculados ya que la herramienta no permite dejar la formula expresada par ser calcula en tiempo de ejecución de la regla, ni la utilización de variables.

Eventos Entrantes

particle

Eventos Salientes

new_particle

Vecindad

La vecindad de las celdas es la vecindad de Moore como se puede observar en la figura 6.

(-1,-1)	(-1,0)	(-1,1)
(0,-1)	(0,0)	(0,1)
(1,-1)	(1,0)	(1,1)

Figura 6: Vecindad de Moore para la celda $(0,0)$

Estados de las celdas

La función *Random* genera, cada vez que es llamada, un número real al azar comprendido en el rango $(0,1)$. Este valor sirve para saber hacia donde debe moverse la partícula. Las celdas pueden tener como estado valores 0, 1 o $(2 + random)$. Inicialmente la matriz toda esta inicializa en estado 0.

- El valor 0 indica que es una celda vacía (sin partícula alguna).
- El valor 1 indica que es una celda con una partícula pegada.
- El valor $(2 + random)$ indica que es una celda con una partícula en movimiento.

Distribución de los eventos de entrada a las celda

Al llegar el evento *particle* al modelo BRW, éste lo distribuye a todas las celdas de la primer fila, llamando a la regla de transición *new_particle_position*. Ver figura 7

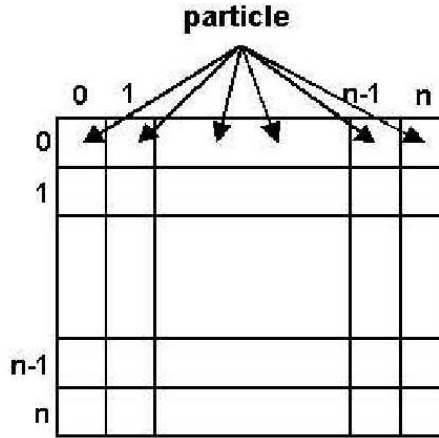


Figura 7: Distribución del evento *particle*

Las siguientes reglas pertenecientes a la transición *new_particle_position* solo hacen que la celda cuya posición (columna) sea igual al número azar del evento pase a tener valor $(2 + \text{random})$. Si la celda no tiene el mismo valor del número al azar del evento, no se modifica y queda con su valor anterior.

[new_particle_position]

```
rule : { 2 + random } 20 { (portValue(thisPort) = CELLPOS(1)) }
rule : { (0,0) } 20 { t }
```

Después de ubicar la partícula en una celda hay que moverla. Cada vez que se mueve la partícula hay que controlar si ésta se pega o no. Si se pega genera un evento de nueva partícula. Ver figura 7.

A continuación se explicarán las reglas por las cuales se mueve y pega una partícula.

[brw-rules]

```
rule : { 1 + send(new_particle,1) } 20 { (trunc((0,0)) = 2) AND
((CELLPOS(0) = 15) OR ( ((-1,-1) = 1) OR ((-1,0) = 1) OR ((-1,1) =
1) OR ((0,-1) = 1) OR ((0,1) = 1) OR ((1,-1) = 1) OR ((1,0) = 1)
OR ((1,1) = 1) ) ) }
```

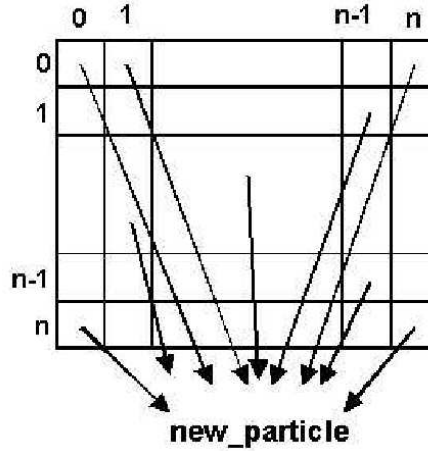


Figura 8: Eventos de salida de la matriz

Esta regla cambia el valor de una celda de $(2 + \text{random})$ a 1, es decir, la *pega* si es una celda con valor entero 2 y está en la última fila o tiene algún vecino inmediato en una vecindad de Moore con valor 1.

```
rule : { 2 + random } 20 { (trunc((0,0)) = 2) AND (CELLPOS(0) = 0)
AND (0 <= fractional((0,0))) AND (fractional((0,0)) < 0.125) }
```

```
rule : { 2 + random } 20 { (trunc((0,0)) = 2) AND (CELLPOS(1) = 0)
AND (0.125 <= fractional((0,0))) AND (fractional((0,0)) < 0.25) }
```

```
rule : { 2 + random } 20 { (trunc((0,0)) = 2) AND (CELLPOS(1) =
15) AND (0.25 <= fractional((0,0))) AND (fractional((0,0)) <
0.375) }
```

Estas tres reglas dicen que si la partícula choca contra el techo o una pared lateral, se queda en el lugar. Por ejemplo si la celda tiene valor entero 2, está en el techo, y el azar (la parte decimal del valor de la celda) dice que debe moverse la partícula hacia arriba, la partícula rebota y queda en su lugar calculando su nuevo valor de $(2 + \text{random})$. Lo mismo para las paredes laterales (si la celda tiene valor entero 2 y está en una pared lateral izquierda o derecha y el azar (la parte decimal del valor de la celda) dice que debe moverse la partícula a izquierda o derecha respectivamente la partícula rebota y queda en su lugar calculando su nuevo valor de $(2 + \text{random})$).

Las siguientes reglas dicen como se mueve la partícula. A diferencia de las reglas anteriores, la celda a la cual la partícula se debe mover debe tener valor 0 y controlar que realmente esa celda se pegue a su nueva posición.

```
rule : { 1 + send(new_particle,1) } 20 { ((0,0) = 0) AND
(trunc((1,0)) = 2) AND (0 <= fractional((1,0))) AND
(fractional((1,0)) < 0.125) AND (CELLPOS(0) != 14) AND ( ((-1,-1)
= 1) OR ((-1,0) = 1) OR ((-1,1) = 1) OR ((0,-1) = 1) OR ((0,1) =
1) OR ((1,-1) = 1) OR ((1,0) = 1) OR ((1,1) = 1) ) }
```

Esta regla pega la partícula una celda hacia arriba. Si la celda tiene valor 0, la celda de abajo tiene valor entero 2, el azar de la celda de abajo dice que debe moverse la partícula hacia arriba, no estoy en la ante-última fila (es decir, la partícula no está en el piso por lo que no se pegaría en el próximo paso de tiempo) y la nueva posición de la celda que estoy evaluando tiene algún vecino con valor 1 en una vecindad de Moore entonces se cambia el valor de la celda a 1 y se dispara el evento de nueva partícula.

```
rule : { 1 + send(new_particle,1) } 20 { ((0,0) = 0) AND
(trunc((0,1)) = 2) AND (0.125 <= fractional((0,1))) AND
(fractional((0,1)) < 0.25 ) AND (CELLPOS(0) != 15) AND ( ((-1,-1)
= 1) OR ((-1,0) = 1) OR ((-1,1) = 1) OR ((0,-1) = 1) OR ((0,1) =
1) OR ((1,-1) = 1) OR ((1,0) = 1) OR ((1,1) = 1) ) }
```

Esta regla pega la partícula una celda a izquierda. Si la celda tiene valor 0, la celda a derecha tiene valor entero 2, el azar de la celda a derecha dice que debe moverse la partícula hacia la izquierda, no estoy en la última fila (es decir, la partícula no está en el piso por lo que no se pegaría en el próximo paso de tiempo) y la nueva posición de la celda que estoy evaluando tiene algún vecino con valor 1 en una vecindad de Moore entonces se cambia el valor de la celda a 1 y se dispara el evento de nueva partícula.

```
rule : { 1 + send(new_particle,1) } 20 { ((0,0) = 0) AND
(trunc((0,-1)) = 2) AND (0.25 <= fractional((0,-1))) AND
(fractional((0,-1)) < 0.375) AND (CELLPOS(0) != 15) AND ( ((-1,-1)
= 1) OR ((-1,0) = 1) OR ((-1,1) = 1) OR ((0,-1) = 1) OR ((0,1) =
1) OR ((1,-1) = 1) OR ((1,0) = 1) OR ((1,1) = 1) ) }
```

Esta regla pega la partícula una celda a derecha. Si la celda tiene valor 0, la celda a izquierda tiene valor entero 2, el azar de la celda a izquierda dice que debe moverse la partícula hacia la derecha, no estoy en la última fila (es decir, la partícula no está en el piso porque sino se pegaría en el próximo paso de tiempo) y la nueva posición de la celda que estoy evaluando tiene algún vecino con valor 1 en una vecindad de Moore entonces se cambia el valor de la celda a 1 y se dispara el evento de nueva partícula.

```
rule : { 1 + send(new_particle,1) } 20 { ((0,0) = 0) AND
(trunc((-1,0)) = 2) AND (0.375 <= fractional((-1,0))) AND
(fractional((-1,0)) < 1 ) AND ( ((-1,-1) = 1) OR ((-1,0) = 1) OR
((-1,1) = 1) OR ((0,-1) = 1) OR ((0,1) = 1) OR ((1,-1) = 1) OR
((1,0) = 1) OR ((1,1) = 1) ) }
```

Esta regla pega la partícula una celda hacia abajo. Si la celda tiene valor 0, la celda de arriba tiene valor entero 2, el azar de la celda de arriba dice que debe moverse la partícula hacia abajo, la nueva posición de la partícula (la celda que estoy evaluando) tiene algún vecino con valor 1 en una vecindad de Moore entonces se cambia el valor de la celda a 1 y se dispara el evento de nueva partícula.

Las siguientes reglas dicen como se mueve la partícula. A diferencia de las reglas anteriores, la celda a la cual la partícula se debe mover debe tener valor 0 y controlar que realmente esa celda se puede mover libremente y no que se pegue en el próximo paso de tiempo.

```
rule : { 2 + random } 20 { ((0,0) = 0) AND (trunc((1,0)) = 2) AND
(0 <= fractional((1,0))) AND (fractional((1,0)) < 0.125) AND
(CELLPOS(0) != 14) }
```

Esta regla mueve la partícula una celda hacia arriba. Si la celda tiene valor 0, la celda de abajo tiene valor entero 2, el azar de la celda de abajo dice que debe moverse la partícula hacia arriba, no estoy en la ante-última fila (es decir, la partícula no está en el piso por lo que no se pegaría en el próximo paso de tiempo) entonces se cambia el valor de la celda a $(2 + random)$.

```
rule : { 2 + random } 20 { ((0,0) = 0) AND (trunc((0,1)) = 2) AND
(0.125 <= fractional((0,1))) AND (fractional((0,1)) < 0.25 ) AND
(CELLPOS(0) != 15) }
```

Esta regla mueve la partícula una celda hacia la derecha. Si la celda tiene valor 0, la celda a izquierda tiene valor entero 2, el azar de la celda a izquierda dice que debe moverse la partícula hacia la derecha, no estoy en la última fila (es decir, la partícula no está en el piso por lo que no se pegaría en el próximo paso de tiempo) entonces se cambia el valor de la celda a $(2 + random)$.

```
rule : { 2 + random } 20 { ((0,0) = 0) AND (trunc((0,-1)) = 2) AND
(0.25 <= fractional((0,-1))) AND (fractional((0,-1)) < 0.375) AND
(CELLPOS(0) != 15) }
```

Esta regla mueve la partícula una celda hacia la izquierda. Si la celda tiene valor 0, la celda a derecha tiene valor entero 2, el azar de la celda a derecha dice que debe moverse la partícula hacia la izquierda, no estoy en la última fila (es decir, la partícula no está en el piso por lo que no se pegaría en el próximo paso de tiempo) entonces se cambia el valor de la celda a $(2 + random)$.

```
rule : { 2 + random } 20 { ((0,0) = 0) AND (trunc((-1,0)) = 2) AND
(0.375 <= fractional((-1,0))) AND (fractional((-1,0)) < 1 ) }
```

Esta regla mueve la partícula una celda hacia abajo. Si la celda tiene valor 0, la celda de arriba tiene valor entero 2, el azar de la celda de arriba dice que debe moverse la partícula hacia abajo entonces se cambia el valor de la celda a $(2 + random)$.

```
rule : { 0 } 20 { (trunc((0,0)) = 2) }
```

Esta regla limpia la celda con valor entero 2 y la deja en 0. Hay que tener en cuenta que si llega a valer esta regla es porque no hay ninguna celda con valor 1 en una vecindad de Moore y no se está en el piso.

```
rule : { (0,0) } 20 { t }
```

Esta regla deja las celdas como estaban originalmente, esta regla se ejecuta siempre que una celda tenga valor 1 o no hayan valido ninguna de las reglas anterior.

Tie-breaking

Dado que solo hay un evento entrante y uno solo saliente, la función de tie-breaking es trivial.

10. Especificación del modelo acoplado

A continuación se especificará el nuevo modelo acoplado. Si bien no se realizará una especificación formal como la dada en clase, se utilizará una especificación alternativa para una fácil y comprensible lectura.

Primeramente se declaran los *eventos entrantes* y *salientes*. Luego, los *componentes* se declaran como nombre del componente y entre paréntesis su tipo. Por último, se declara la forma en que los componentes se acoplan, *acoplamiento de eventos entrantes*, *salientes* e *internos*. Ver figura 5.

Eventos Entrantes

```
start
```

Eventos Salientes

Componentes

```
gen_particle (GeneradorParticulas), brw (brw)
```

Acoplamiento de eventos entrantes

```
start --> gen_particle.new_particle
```

Acoplamiento interno

```
gen_particle.particle --> brw.particle gen_rnd.random -->
brw.random brw.new_particle --> gen_particle.new_particle
```

Acoplamiento de eventos salientes

No existen acoplamiento de eventos salientes.

Tie-breaking

El orden de prioridad de los eventos es el siguiente:

1. *start*
2. *particle*
3. *new_particle*

11. Resultados

Los resultados observados son iguales al modelo del *autómata celular*. El nuevo modelo presenta el mismo comportamiento, aunque la performance general ha sido mejorada.

12. Conclusión

Sobre la primer parte del trabajo se obtienen las siguientes conclusiones:

Si bien el modelo elegido no utiliza todo el potencial otorgado por CELL-DEVS, sino que más bien se comporta como un autómata celular, este trabajo sirvió para comprender mejor aún el uso de la herramienta.

Al observar los archivos de modelo “.ma”, puede verse que los éstos se vuelven cada vez más extensos a medida que el tamaño de la matriz crece. Tubo que crearse un pequeño programa para generar todos los *links* necesarios del modelo *brw*. Esto se podría haber reducido utilizando las características de macro que ofrece la herramienta. Queda como trabajo para futuro el uso de macros para disminuir el tamaño de los archivos de modelo “.ma”.

Se realizo una prueba cambiando el tipo de los modelos de *CELL* a *FLAT*. Esto resulto en una mejora en el tiempo de simulación. Sin embargo, comparando éste modelo con una aplicación realizada en *C* que implementa el mismo modelo de simulación, el tiempo de simulación es de varios orden menor. Claro está que la aplicación desarrollada solo sirve para realizar la simulación de este modelo y cualquier modificación al modelo requerirá de una modificación al código fuente con su consecuente compilación y linkediación. La herramienta no sufre de este problema, lo que la hace más sencilla de utilizar, la puede usar cualquier persona con del modelo y como puede ser expresado con las reglas de modelaje de la herramienta, no se necesita ser un experto en *programación C*.

Podemos sacar en conclusión que la herramienta CELL-DEVS resulta conveniente para expresar rápidamente modelos complejos en ambientes pequeños, y que el desarrollo de aplicación específicas de simulación son conveniente cuando el modelo no muy complejos que necesita ambientes amplios. Por “ambiente” me refiero al tamaño de la matriz necesaria par la simulación.

Comparando la segunda parte contra la primera, se observa que el tiempo real de simulación del modelo de la segunda parte es menor que el tiempo del primer modelo. Lo mismo pasa con el archivo de *log*.

Esto se debe a que no existe el componente atómico *GeneradorRandom* que generaba un evento de *random* para cada celda. Esto hizo que se disminuyera la cantidad de mensaje a enviar y recibir. Al no existir el componente *GeneradorRandom*, la creación de los archivos de modelo *.ma* se hizo más sencilla al no haber tantos *links* entre celdas y mensajes desde y hacia *GeneradorRandom*.

Otro cambio se presenta en la vecindad. Ésta disminución ocasiona que se ejecuten menos reglas, y por consiguiente que hayan menos mensajes.

Esta segunda parte es mucho mas eficiente que la primera ya que fue pensado no como un autómata celular, sino como un modelo DEVS. Y al ser pensado como un modelo DEVS, utiliza las herramientas que ofrece este formalismo.

si bien ambas partes son posible soluciones al problema BRW, la segunda es mucha más eficiente que la primera parte por la razones antes mencionadas, pero principalmente por ser un modelo DEVS.

Índice de figuras

1.	BRW	4
2.	Vecindad de la celda $(0, 0)$	6
3.	Distribución del evento <i>particle</i>	7
4.	Distribución del evento <i>random</i>	8
5.	Nuevo modelo BRW	14
6.	Vecindad de Moore para la celda $(0, 0)$	15
7.	Distribución del evento <i>particle</i>	16
8.	Eventos de salida de la matriz	17

Referencias

- [1] <http://apricot.polyu.edu.hk/~lam/dla/dla.html>
- [2] <http://www.oche.de/~ecotopia/dla/>
- [3] <http://www.stats.bris.ac.uk/~maxsv/PAPERS/brw.ps.gz>
- [4] <http://www.ime.usp.br/~popov/ps/mmp.ps.gz>