SystemVerilog pour la vérification

Randomisation

YTA

Reconfigurable and Embedded Systems Institute
Haute Ecole d'Ingénierie et de Gestion du canton de Vaud

Octobre 2010

Plan

Génération aléatoire

- La vérification basée sur l'aléatoire a besoin de ...
- Génération aléatoire
- VHDL ne possède pas de telle possibilité (accessible via des librairies)
- SystemVerilog si

Génération aléatoire

Fonctions standards

```
// Retourne un nombre aléatoire
// La racine peut être passée en paramètre
function int unsigned $urandom[(int seed)];

// Retourne un nombre aléatoire entre deux bornes
function int unsigned $urandom_range(int unsigned maxval,
                                     int unsigned minval=0);

// Initialise le générateur pseudo-aléatoire
// Peut être appliqué à un objet ou un thread
function void $srandom(int seed);
```

Génération aléatoire

Exemple

```
module top_random;

    initial begin
        int A;
        $srandom(5);                  // Init. du générateur du thread
        A = $urandom;                 // Randomisation de la variable
        $display(A);
        A = $urandom_range(100,0); // Randomisation de la variable
        $display(A);
    end

endmodule
```


Randomisation de variables

- Ces fonctions de base sont intéressantes, mais limitées
- SystemVerilog offre un mécanisme plus puissant:
  - La randomisation des objets!
- Une variable (dans un classe) peut être déclarée *rand*:

Exemple

```
class Packet;
    rand bit [7:0]  m_address;
    rand bit [31:0] m_data;
endclass : Packet
```

Randomisation d'un object

- Un objet entier peut être randomisé
- La randomisation est appliquée sur toutes les variables rand

Exemple

```
Packet P;
initial begin
    P = new();
    if (!P.randomize()) $stop;
    $display(P.m_address);      // contient une valeur aléatoire
    $display(P.m_data);         // contient une valeur aléatoire
end

assert (P.randomize()) else $error("No solutions for P.randomize");
```


Randomisation contrainte

- L'espace des possibles peut être immense
- Toutes les combinaisons ne sont pas forcément pertinentes
- Il est possible de contraindre la randomisation

Exemple

```
class Packet;
    rand bit [7:0]  m_address;
    rand bit [31:0] m_data;
    constraint addr_range {
        m_address < 132;
    }
endclass : Packet
```

Ajout de contraintes par héritage

- Il est possible d'ajouter des contraintes dans une sous-classe

```
class Word_Packet extends Packet;
constraint word_align { m_address[1:0] == `0;}
endclass : Word_Packet
```


Modification dynamique de contraintes

- Les contraintes peuvent se référer à des variables non aléatoires
- La randomisation peut dépendre de ces variables

```
typedef bit [7:0] addr_t;
typedef bit [31:0] data_t;
class Packet;
    rand addr_t m_address;
    rand data_t m_data;
    addr_t high_address = `1;
    addr_t low_address = 0;
    constraint addr_range {
        m_address <= high_address;
        m_address >= low_address;
    }
endclass : Packet
```

Modification dynamique de contraintes

```
Packet P;
initial begin
    P = new();
    P.high_address = 10;
    assert (P.randomize) else $stop; // range is 0-10
    $display(P.m_address);
    P.high_address = '1;
    P.low_address = 250;
    assert (P.randomize) else $stop; // range is 250-255
    $display(P.m_address);
end
```


Over Constraining

```
class Packet;
    rand bit [7:0] m_address;
    rand bit [31:0] m_data;
    rand bit m_data_parity;
    constraint addr_range {
        m_address < 132;
    }
    constraint gen_data_parity {
        m_data_parity == even_parity(m_data);
    }
    function bit even_parity(bit [31:0] d);
        return (~^d);
    endfunction
endclass : Packet
```

- Est-ce une bonne solution?
- Si une solution aux contraintes existe, le solver la trouvera
  - Mais: le processus peut consommer beaucoup de temps CPU
  - Alors: les corrélations peuvent être exploitées

Over Constraining

- Solution: utiliser *post_randomize()*
- Fonction appelée automatiquement après la randomisation

```
class Packet;
    rand bit [7:0] m_address;
    rand bit [31:0] m_data;
    bit m_data_parity;
    constraint addr_range {
        m_address < 132;
    }
    function void post_randomize();
        m_data_parity = ~^m_data;
    endfunction
endclass : Packet
```


Pré-post traitement

- Pré-traitement: *pre_randomize()*
  - Appelée avant la randomisation
  - Peut servir à allouer un tableau dynamique (p. exemple)
- Post-traitement: *post_randomize()*
  - Appelée après la randomisation
  - Peut servir à calculer des CRCs (p. exemple)
  - Peut permettre d'injecter des erreurs

Pré-post traitement

Corps des fonctions

```
function void pre_randomize();
    if (super)
        super.pre_randomize();
    // Placer ici le traitement à effectuer
endfunction

function void post_randomize();
    if (super)
        super.post_randomize();
    // Placer ici le traitement à effectuer
endfunction
```

- L'appel à la méthode de la super-classe est nécessaire pour appliquer le traitement aux variables de la super-classe

Implication

- Contrainte de type "si A, alors B" (A->B)
- But: Appliquer la contrainte B si l'expression A est vraie

```
typedef bit [7:0] addr_t;
typedef enum {READ,WRITE,NOP} kind;

class Packet;
    rand addr_t m_address;
    rand bit [31:0] m_data;
    rand kind m_op;
    constraint data_range {
        (m_op == READ) -> m_data inside {[1:100]};
        (m_op == WRITE) -> m_data inside {[101:255]};
        (m_op == NOP) -> m_data inside {0};
    }
endclass : Packet
```

Ordre de résolution

- Il est possible de définir l'ordre de résolution des variables

```
class Packet;
    rand addr_t m_address;
    rand bit [31:0] m_data;
    rand kind m_op;
    constraint data_range {
        (m_op == READ) -> m_data inside {[1:100]};
        (m_op == WRITE) -> m_data inside {[101:300]};
        (m_op == NOP) -> m_data inside {0};
    }
    constraint order {solve m_op before m_data;}
endclass : Packet
```


Distributions de probabilités

- Il est possible d'ajouter un poids aux différentes valeurs possibles

```
class Packet;
    rand addr_t m_address;
    rand bit [31:0] m_data;
    rand kind m_op;
    constraint data_range {
        (m_op == READ) -> m_data inside {[1:100]};
        (m_op == WRITE) -> m_data inside {[101:300]};
        (m_op == NOP) -> m_data inside {0};
    }
    constraint op_dist { m_op dist {READ := 2, WRITE := 2 NOP := 1};}
endclass : Packet
```

Injection d'erreur

- Exemple: Injection d'erreurs dans la parité

```
class Packet;
  rand bit [7:0] m_address;
  rand bit [31:0] m_data;
  bit m_data_parity;

  constraint addr_range {
    m_address < 132;
  }

  function void post_randomize();
    m_data_parity = ~^m_data;
  endfunction
endclass : Packet
```

```
class PacketParityError extends Packet;

  rand bit m_error;

  constraint error_dist {
    m_error dist {1b0:=99, 1b1:=1};}

  function void post_randomize();
    super.post_randomize();
    if (m_error)
      m_data_parity = ~m_date_parity;
  endfunction
endclass : Packet
```


Génération aléatoire: initialisation

- Un générateur pseudo-aléatoire est associé à chaque
  - Thread
  - Objet

```
module top_random;
    class Test_random;
        int A;
    endclass

    initial begin
        int module_A;
        Test_random my_obj;
        my_obj=new();                  // Création de l'objet
        process::self.srandom(5);      // Init. du générateur du thread
        assert(randomize(module_A));   // Randomisation de la variable
        my_obj.srandom(100);           // Init. du générateur de l'objet
        assert(my_obj.randomize());    // Randomisation de l'objet
        module_A = $urandom;
        $display(A);
    end
endmodule
```