SystemVerilog pour la verification

Patrons de conception et introduction à UVM

Tarik Graba
tarik.graba@telecom-paristech.fr
Année 2019/2020

Plan

Patrons de Conception (Design patterns)

Introduction à UVM

Développement orienté objet

Pourquoi ?

- Séparé le *testbench* du *design*.
- Permettre la modularité et la réutilisation de code.
- Modification dynamique du testbench (durant la simulation)
- Modèles éprouvés dans le monde du développement Logiciel.

Pré-requis

Paramétrisation

- Une classe peut avoir des paramètres.
  - type ou valeur
- Définis avec **#**.
- Peuvent avoir des valeurs par défaut.
- S'il n'en n'ont pas, il est obligatoire de les définir à l'instanciation.

```
program test;

class vect #(type T=bit, int W = 4);
   T[W-1:0] v;
endclass:vect

vect V;
vect#(logic, 16) W;

initial
begin
   V = new();
   $display("%p",V);
   $display("%h",V.v);
   $display("%p",$typename(V));

   W = new();
   $display("%p",W);
   $display("%h",W.v);
   $display("%p",$typename(W));

end


endprogram
```

Pré-requis

Champs statiques

```
program test;

class Obj;
    int id;
    static int count = 0;
    function new;
       id = count++;
    endfunction
endclass:Obj

Obj o;

initial
begin
   $display("1- %p",o);
   $display("2- %d",o.count);
   $display("3- %d",Obj::count);
   repeat(5)
   begin
      o = new();
      $display("+- %p",o);
   end
   $display("2- %d",o.count);
   $display("3- %d",Obj::count);
end

endprogram
```

- Une classe peut avoir des champs ou des méthodes statiques.
  - Une seule instance pour tous les objets.
- Les instances statiques sont normalement créées avant le début de la simulation.

Pré-requis

Champs statiques

```
program test;

class Obj;
    int id;
    static int count = 0;
    function new;
       id = count;
       count++;
    endfunction
endclass:Obj

initial
begin
   $display(">- %d",Obj::count);
end


endprogram
```

- Une classe peut avoir des champs ou des méthodes statiques.
  - Une seule instance pour tous les objets.
- Les instances statiques sont normalement créées avant le début de la simulation.

Pré-requis

Champs statiques d'une classe paramétrée

```
program test;

class Obj #(type T);
    static T sval = '1;
endclass:Obj

typedef Obj#(bit[3:0]) static_type_4;
typedef Obj#(bit[7:0]) static_type_8;

initial
begin
   $display(">- %d",static_type_4::sval);
   $display(">- %d",static_type_8::sval);
end


endprogram
```

- Si on ne connait pas la valeur des paramètres d'une classe, ses champs statiques ne sont pas créés.
- En SystemVerilog, un **`typedef`** suffit pour créer les champs statiques.

Singleton

Référence vers un objet unique

- Permet de créer d'obtenir une référence vers un objet unique.
- Une classe avec un constructeur privé (**`local`**) et une méthode **`get`** pour renvoyer une référence vers un instance statique de la classe.

```
program test;

class Racine;

    local static Racine m_racine;

    // le constructeur est local (privé)
    local function new();
    endfunction

    // Fonction statique, il n'en n'existe qu'une
    // renvoi une référence vers une instance statique
    static function Racine get();
       if(m_racine == null) m_racine = new();
       return m_racine;
    endfunction

endclass:Racine

....
```

Singleton

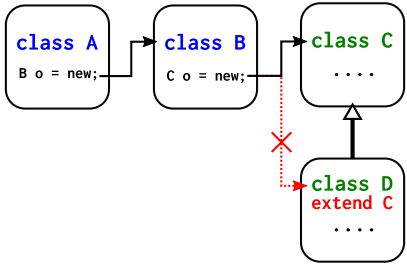
Référence vers un objet unique

- Permet de créer d'obtenir une référence vers un objet unique.
- Une classe avec un constructeur privé (**`local`**) et une méthode **`get`** pour renvoyer une référence vers un instance statique de la classe.

```
...
class Obj;
   Racine t;
   function new (Racine t);
      this.t = t;
   endfunction
endclass

Racine top;
Obj o;

initial
begin
   // Appel illégal au constructeur privé
   // top = new();

   // ref ver le singleton
   top = Racine::get();
   // o est construit avec une référence vers
   // le singleton
   o = new(Racine::get());
   $display("%p",top);
   $display("%p",Racine::get());
   $display("%p",o);
end

endprogram
```

Factory

Pourquoi ?



- On veut pouvoir décider le plus tard possible du type de l'objet construit.
- Impossible avec le constructeur (**new**)
- Déléguer la construction à objet qui prendra la décision en fonction du contexte (La **Factory**)

Factory

Comment ? Une classe proxy

```
program test;

//  interfaces
virtual class Obj;
   pure virtual function void dump();
endclass

virtual class ObjProxy;
   pure virtual function Obj createObj();
endclass

// Les vraie classes
class A extends Obj;
   int a;
   virtual function void dump();
      $display("-A- %p", this);
   endfunction
endclass

class ObjProxyA extends ObjProxy;
   virtual function Obj createObj();
      A a;
      a = new();
      return a;
   endfunction
endclass
...
```

- On n'appelle pas le constructeur directement
- On demande au proxy de construire l'instance pour nous
- On doit le caster si on veut un objet du type fils

Factory

Comment ? Une classe proxy

```
...
initial
begin
   ObjProxy p;
   Obj o;
   A a;

   // Enregistrement de la factory
   p = ObjProxyA::new();

   o = p.createObj();
   o.dump();

   $cast(a , p.createObj());
   a.dump();
end

endprogram
```

- On n'appelle pas le constructeur directement
- On demande au proxy de construire l'instance pour nous
- On doit le caster si on veut un objet du type fils

Factory

Comment ? 2 classes proxy

```
...
class B extends A;
   int b;
   virtual function void dump();
      $display("-B- %p", this);
   endfunction
endclass

class ObjProxyB extends ObjProxy;
   virtual function Obj createObj();
      B b;
      b = new();
      return b;
   endfunction
endclass
....
```

- Si on ajoute une classe, on ajoute un proxy
- Généralement on enregistre les proxy dans une base de données.
  - Dans l'exemple un tableau dynamique ou une table de hachage.

Factory

Comment ? 2 classes proxy

```
....
initial
begin
   ...
   // dynamic array (factory database)
   ObjProxy p[$];
   // associative array (factory database)
   ObjProxy F[string];

   // Enregistrement des factorys
   p[0] = ObjProxyA::new();
   p[1] = ObjProxyB::new();

   foreach ( p[i] ) begin
      o = p[i].createObj(); o.dump();
   end

   foreach ( p[i] ) begin
      $cast(a,  p[i].createObj()); a.dump();
   end

   // Modification dynamique de la factory
   F["A"] = ObjProxyA::new();
   $cast(a , F["A"].createObj()); a.dump();
   F["A"] = ObjProxyB::new();
   $cast(a , F["A"].createObj()); a.dump();
end
endprogram
```

- Si on ajoute une classe, on ajoute un proxy
- Généralement on enregistre les proxy dans une base de données.
  - Dans l'exemple un tableau dynamique ou une table de hachage.

Factory

Comment ? Une classe proxy générique

- Une classe proxy paramétrée par le type d'objet à construire

```
...
class ObjProxyT #(type T) extends ObjProxy;
   virtual function Obj createObj();
      T r;
      r = new();
      return r;
   endfunction
endclass
...
```

Factory

Comment ? Une classe proxy générique

- Une classe proxy paramétrée par le type d'objet à construire

```
....
   // Enregistrement de la factory
   p[0] = ObjProxyT#(A)::new();
   p[1] = ObjProxyT#(B)::new();

...
   // Modification de la factory
   F["A"] = ObjProxyT#(A)::new();
   $cast(a , F["A"].createObj()); a.dump();
   F["A"] = ObjProxyT#(B)::new();
   $cast(a , F["A"].createObj()); a.dump();
...
```

Factory

Comment ? Une classe proxy générique

- Une classe proxy paramétrée par le type d'objet à construire
- L'enregistrement est manuel…
- On est obligé de faire un cast…

```
....
   // Enregistrement de la factory
   p[0] = ObjProxyT#(A)::new();
   p[1] = ObjProxyT#(B)::new();

...
   // Modification de la factory
   F["A"] = ObjProxyT#(A)::new();
   $cast(a , F["A"].createObj()); a.dump();
   F["A"] = ObjProxyT#(B)::new();
   $cast(a , F["A"].createObj()); a.dump();
...
```

Factory

Comment ? Une classe qui enregistre les proxy

```
...
// associative array (factory database)
ObjProxy FactoryDB[ObjProxy];

class ObjRegistry#(type T) extends ObjProxy;
   // La méthode qui crée vraiment un objet
   virtual function Obj createObj();
      T r = new();
      return r;
   endfunction
   // Singleton
   local static ObjRegistry#(T) M = getFact();

   static function ObjRegistry#(T) getFact();
      if( M == null) begin
         M = new();
         // Enregistrement
         FactoryDB[M] = M;
      end
      return M;
   endfunction:getFact
   // `create` la méthode qu'on appelle
   static function T create();
      T o;
      $cast(o, FactoryDB[getFact()].createObj());
      return o;
   endfunction:create
endclass:ObjRegistry
...
```

- Un singleton paramétrable
- Enregistre automatiquement la factory dans une table de hachage
- Pas besoin de l'instancier, juste un **`typedef`** dans la classe qu'on crée.

Factory

Comment ? Une classe qui enregistre les proxy

```
...
// Les vraies classes
class A extends Obj;

   // Créera un proxy et l'enregistrera  pour A
   typedef ObjRegistry#(A) typeId;

   int a;
   virtual function void dump();
      $display("-A- %p", this);
   endfunction
endclass

class B extends A;
   typedef ObjRegistry#(B) typeId;
   int b;
   virtual function void dump();
      $display("-B- %p", this);
   endfunction
endclass
...
```

- Un singleton paramétrable
- Enregistre automatiquement la factory dans une table de hachage
- Pas besoin de l'instancier, juste un **`typedef`** dans la classe qu'on crée.

Factory

Comment ? Une classe qui enregistre les proxy

```
...
initial
begin
   A a;
   B b;

   // typeId fait référence à ObjRegistry#(A)
   a = A::typeId::create();
   a.dump();
   $display("**> %p",$typename(a));

   // Échouera
   //$cast (b, a);

   // Modification de la factory
   FactoryDB[A::typeId::getFact()] =
                          B::typeId::getFact();

   // create de A appelle maintenant create de B
   a = A::typeId::create();
   a.dump();
   $display("**> %p",$typename(a));

   $cast (b, a);
   b.b = 33;
   b.dump();
   $display("**> %p",$typename(b));
end
...
```

- Un singleton paramétrable
- Enregistre automatiquement la factory dans une table de hachage
- Pas besoin de l'instancier, juste un **`typedef`** dans la classe qu'on crée.

Plan

Patrons de Conception (Design patterns)

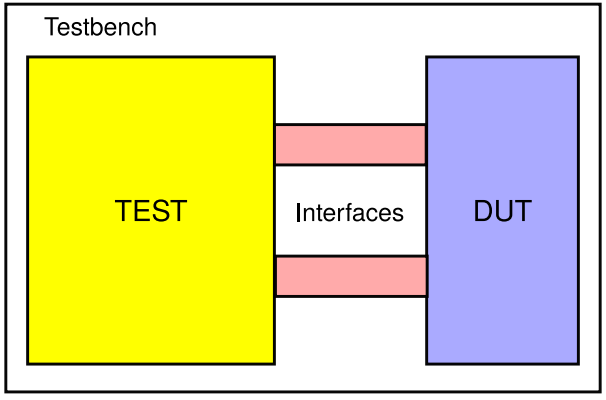
Introduction à UVM

UVM c'est quoi ?

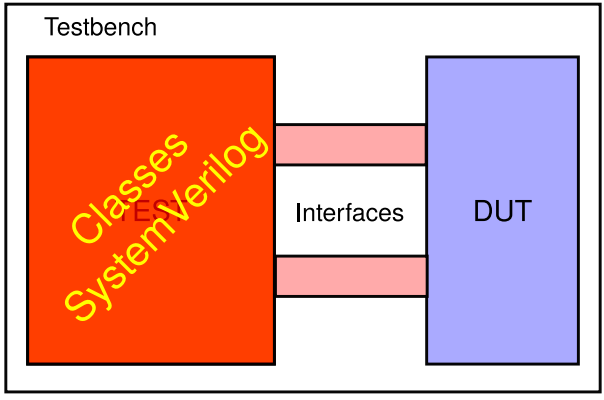
Universal Verification Methodology

- Une méthodologie de vérification
  - Utilisant le modèle objet de SystemVerilog
- Un standard industriel maintenu par Accellera
  - `http://www.accellera.org/community/uvm/`
- Un ensemble de classes de base
  - Le code est distribué sous les termes de la licence Apache

UVM c'est quoi ?



UVM c'est quoi ?



UVM c'est quoi ?

Hiérarchie d'un testbench UVM

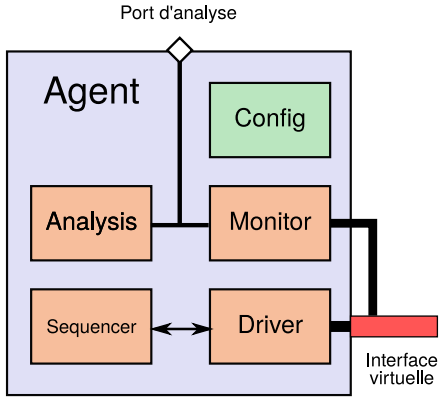
UVM définit plusieurs structures permettant de hiérarchiser un testbench. Les deux principales sont :

- L'agent (`Agent`)
  - Contrôle et observe l'interface
  - Reçoit et séquence les transactions
  - Permet l'analyse
- L'environnement (`Env`)
  - Contient les configurations des différents sous-blocks
  - Définit la structure du test (contient des agents par exemple)
  - Contient les éléments nécessaires à la validation du test

Ce sont des classes et pas des modules !

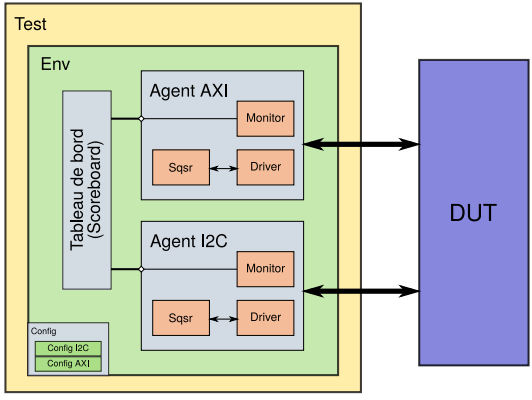
UVM c'est quoi ?

L'agent



UVM c'est quoi ?

L'environnement



UVM c'est quoi ?

Les classes prédéfinie

- Dans la bibliothèque des classes de base sont prédéfinies pour les differents éléments structurels
- Par exemple :

`uvm_env`	pour définir un environnement
`uvm_agent`	pour l'agent
`uvm_test`	pour le test

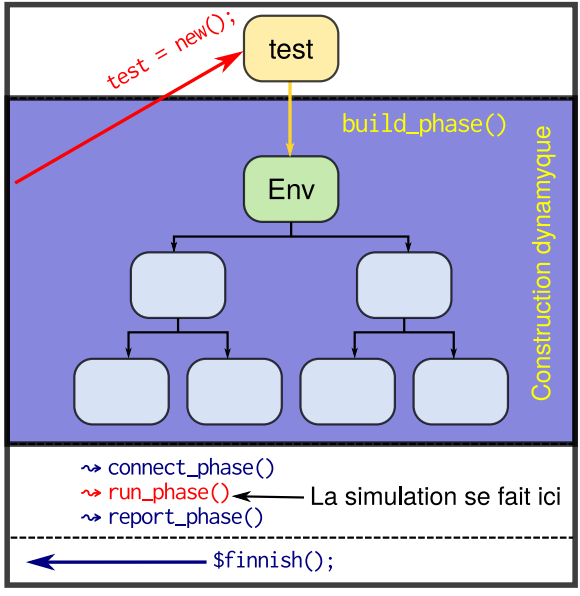
- On spécialise certaines de ces classes en fonction des besoins de test.

Liste des classes dans la documentation officielle.

UVM c'est quoi ?

Comment ça marche

```
module testbench;
import uvm_pkg::*;
import test_pkg::*;

interface_yy yy(...);
DUT xx(...);

initial begin
   run_test();    ---------------->

end               <----------------
endmodule
```



UVM c'est quoi ?

Comment ça marche

- On appelle la fonction (méthode statique) **`uvm_test`**.
  - Un objet de test est créé (**`uvm_test`**).
- La méthode **`build_phase`** est appelée et construit l'environnement en y associant une configuration.
  - Tous les sous éléments de l'environnement sont construits par des appels en cascade à leur méthode **`build_phase`**.
- La simulation commence vraiment quand la méthode **`run_phase`** est appelée.
  - Ce n'est qu'ici que le temps peut avancer.
- À la fin **`$finnish()`** est appelé