Introduction à SystemC

Mixer les langages « La co-simulation »

Tarik Graba

2020-2021

Table des matières

Utiliser SyetemC pour la simulation et la vérification de modèle RTL écrits dans d’autres langages.

Les modèles RTL sont généralement écrits en utilisant les sous-ensembles synthétisables des langages HDL communs (VHDL, Verilog/SystemVerilog)

Même dans ce cas, SystemC peut être utilisé pour concevoir un environnement de simulation pour vérifier et valider le modèle RTL.

Les avantages de l’utilisation de SystemC viennent principalement de :

- C++ permet de structurer/réutiliser l’environnement,
- Langage compilé *efficace*,
- utiliser les bibliothèques systèmes,
- interaction poussée avec le simulateur,
- se passer d’outils propriétaires.

D’un autre côté :

- C++ est un langage complexe pas forcément maitrisé par les concepteurs HDL,
- il existe des alternatives (*DPI/VPI*) pour utiliser des bibliothèques système dans un environnement de simulation en SystemVerilog,
- certaines méthodologies de vérification font référence dans l'industrie et elle n'utilisent pas forcement SyetemC.

---

1 Mixer les langages

La co-simulation

La **co-simulation** consiste en la mise en œuvre d'une simulation dans laquelle les éléments ont été écrits dans des langages différents.

Pour pouvoir le faire :

- les concepts manipulés doivent pouvoir s'exprimer dans les deux langages,
- l'outil (*simulateur*) doit les supporter et permettre de passer de l'un à l'autre.

Si on doit faire une simulation évènementielle, les langages utilisés doivent prévoir les concepts :

- d'évènements,
- de processus.

De plus, on doit pouvoir représenter la structure des *modules* simulés dans les deux langages.

Les langages RTL communément utilisés (VHDL, Verilog/SystemVerilog) ont été conçus autour de ces concepts. SystemC ajoute ces concepts à C++.

De plus, il faut pouvoir exprimer les données manipulées avec la bonne granularité (entier, bits, vecteurs/bus…) dans les deux langages. Il n'est généralement pas possible de faire passer d'un langage vers l'autre des structures de données plus complexes (interfaces SystemVerilog, des structures…).

Généralement le point bloquant vient de l'outil utilisé :

- que supporte-t-il dans le langage A et B,
- que sait-il transmettre de A vers B,
- comment l'indiquer à l'outil ?

Testbench en SystemC

Ce que nous allons faire ici c'est simuler un design HDL décrit en Verilog/SystemVerilog dans un environnement en SystemC.

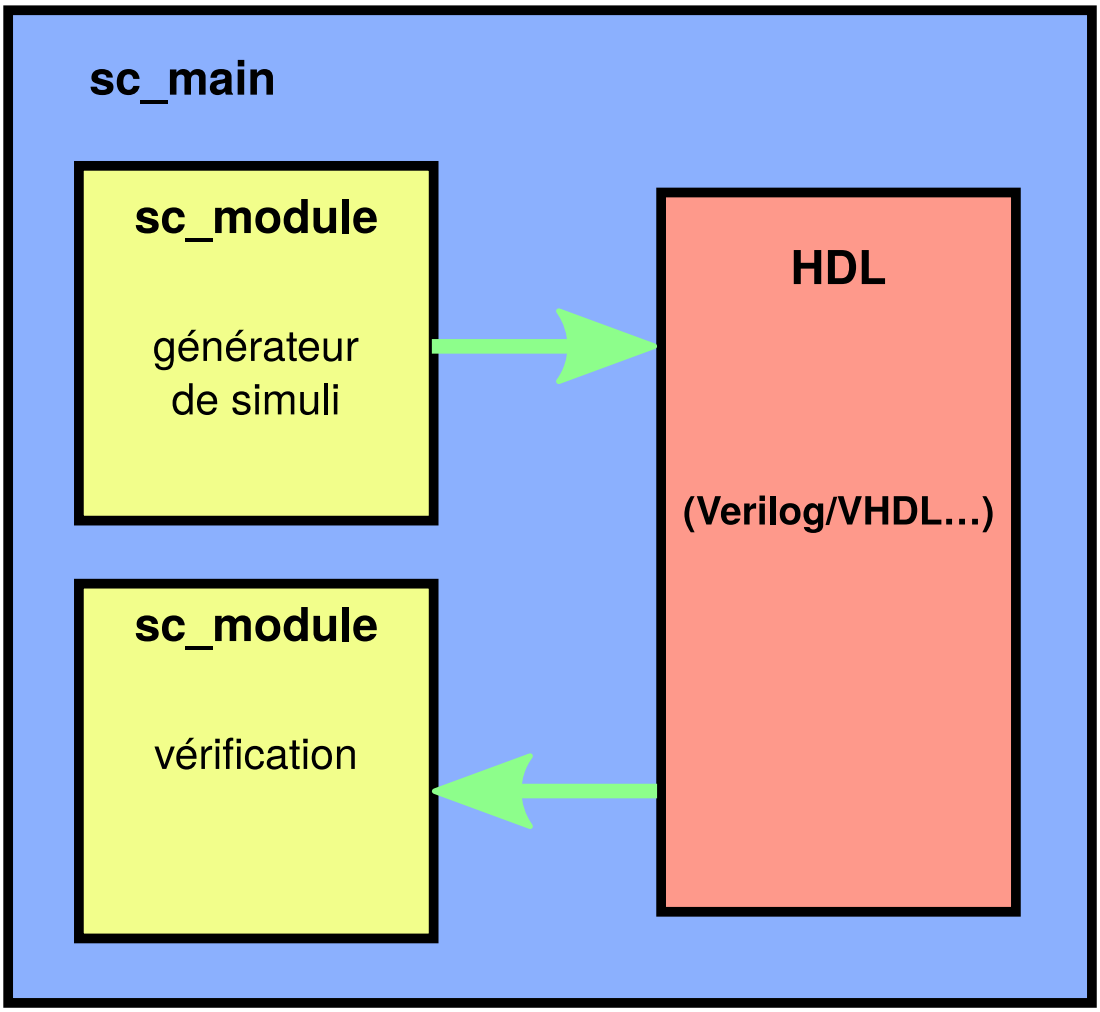

FIG. 1.1: Co-simulation

Cet environnement contiendra de quoi générer des stimuli et vérifier les réponses du bloc HDL (*DUT :* Design Under Test).

Il serait aussi envisageable d'insérer un module SystemC dans un banc de test (*testbench*) écrit dans un autre langage, mais ce ne sera pas présenté dans ce cours.

---

2 Mise en pratique

Deux approches

Nous allons voir deux exemples de simulation d'un module RTL (Verilog/SystemVerilog) dans un testbench écrit en SyetemC.

Deux approches :

- Utiliser un simulateur qui accepte les deux langages (**Modelsim**).
- Utiliser un convertisseur (**Verilator**) qui transforme le RTL dans un langage que l'on peut compiler.

Ici nous allons mettre en œuvre deux approches differentes :

1. En utilisant Modelsim, un simulateur commercial qui supporte l'utilisation de plusieurs langages HDL, nous pouvons compiler séparément les différents fichiers sources. La seule contrainte viendra du fait que le code SystemC étant du C++ il faudra d'une que les modules HDL soient définis d'une scertaine façon au moment de la compilation des fichiers C++ qui y font référence. Le simulateur s'occupera par la suite de faire communiquer l'ensemble de façon transparente.
2. En utilisant Verilator un outil qui permet de générer des modèles C++ et SystemC à partir de modèles en Verilog/SystemVerilog. C'est un outil libre (OpenSource) destiné à générer des modèles de simulation à partir de modèles RTL synthétisables. Le tout est ensuite compilé et le simulateur intégré à la bibliothèque SystemC peut alors être utilisé.

---

3 Modelsim

Module “étranger”

Avec Modelsim on doit utiliser un module *wrapper* SystemC qui est compilé avec le testbench. Ce module est une coquille vide qui ne contient pas de code comportemental, mais doit :

- reprendre l’interface du module HDL (les entrées/sorties) pour permettre de faire les connexions,
- utiliser un mécanisme (`foreign module`) du simulateur pour le lier au module HDL pour mettre en relation le wrapper avec le module HDL durant la phase d’élaboration.

Ce wrapper peut être écrit à la main ou généré automatiquement après compilation du module HDL (`scgenmod --help` pour plus d’informations.)

Exemple (HDL) :

```
module counter ( input clk, nrst, enable,
                 output logic[7:0] Q
               );

// ...

endmodule
```

Exemple (Wrapper) :

```
#ifndef _SCGENMOD_counter_
#define _SCGENMOD_counter_

#include ”systemc.h”

class counter : public sc_foreign_module
{
public:
```

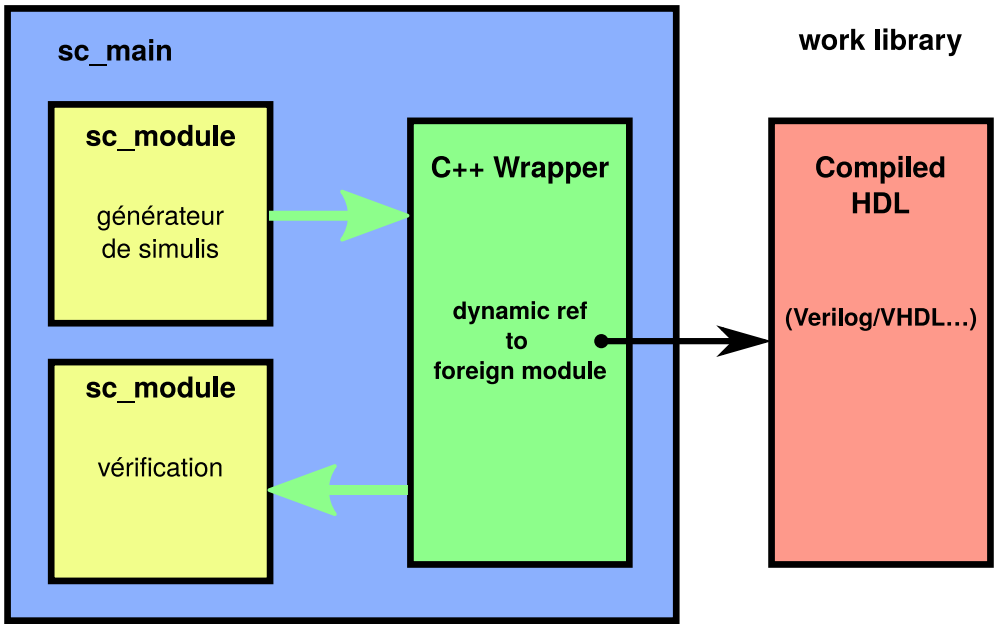

FIG. 3.1: Foreign Module

```
    sc_in<bool> clk;
    sc_in<bool> nrst;
    sc_in<bool> enable;
    sc_out<sc_uint<8> > Q;


    counter(sc_module_name nm, const char* hdl_name)
     : sc_foreign_module(nm),
       clk("clk"),
       nrst("nrst"),
       enable("enable"),
       Q("Q")
    {
        elaborate_foreign_module(hdl_name);
    }
    ~counter()
    {}

};

#endif
```

Contrairement à un `sc_module`, ici on hérite d'un `sc_foreign_module` qui est une classe propre à Modelsim.

L'interface de ce module (ses entrées/sorties) correspondent à celles du module HDL. Les types qui existent dans les deux langages ne sont pas les même, Modelsim (se référer à la documentation de l'outil) prévoit plusieurs possibilités. Dans cet exemple, on fait correspondre aux bits Verilog des `bool` et aux vecteurs des `sc_uint<>` mais nous aurions pu utiliser respectivement des `sc_logic` et des `sc_lv`.

Le constructeur de ce module prend 2 arguments, l'un correspondant au nom de l'instance (`sc_module_name`) au sens SystemC et l'autre correspondant au nom du module HDL auquel il doit être lié. Sinon, le constructeur ne fait qu'appeler une méthode `elaborate_foreign_module` qui déclenchera un mécanisme interne du simulateur pour l'élaboration du module RTL.

Exemple (Instanciation) :

```
#include <systemc.h>
#include "counter.h"

int sc_main(int argc, char * argv[]) {
   //..
   //                inst. name   RTL module
   counter counter_0("counter_0", "counter");
   counter_0.clk( xx );
   //...
   counter counter_1("counter_1", "counter");
   counter_1.clk( xx );
   //...
   return 0;
}
```

Ensuite, le module wrapper peut être instancier et utilisé comme un module SystemC standard.

Workflow `Modelsim`

La génération du module wrapper peut être automatisée en utilisant la commande `scgenmod` de Modelsim après compilation du module HDL. `scgenmod` possède aussi des options pour préciser les correspondances

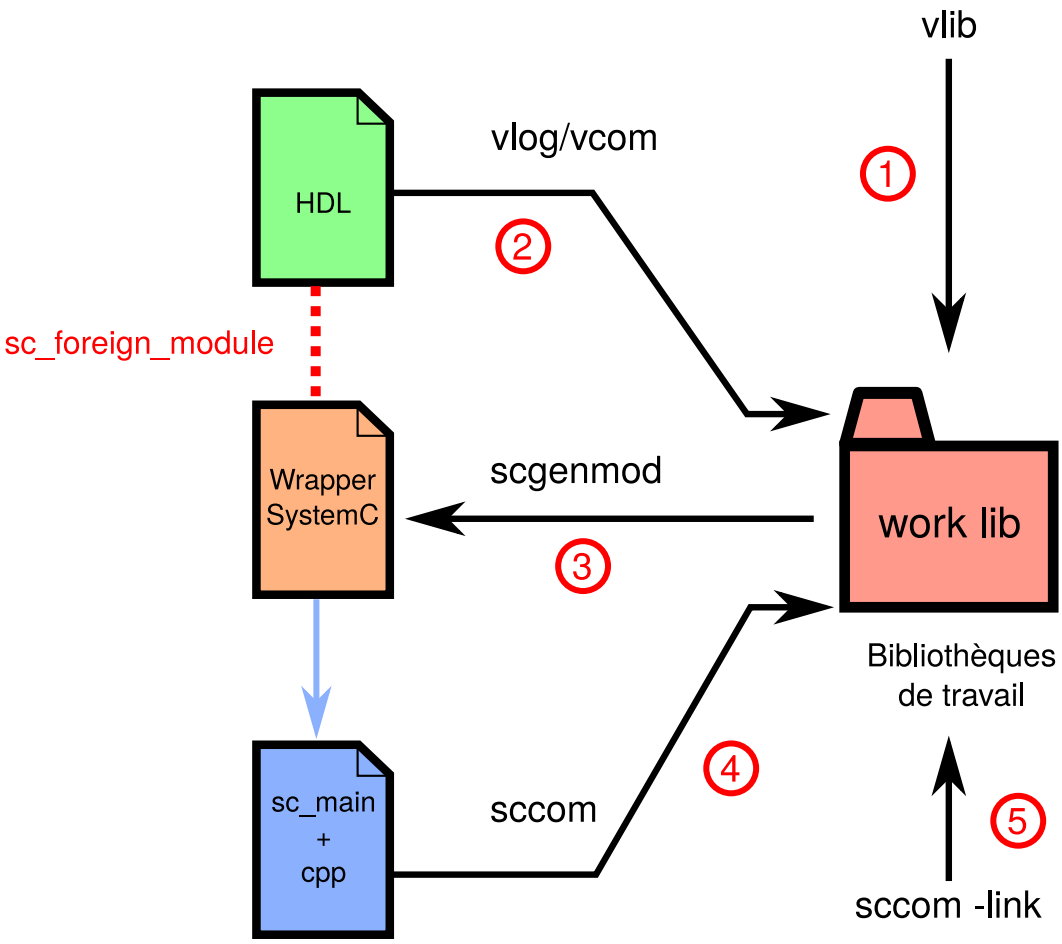

Fig. 3.2: Foreign Module

entre les types HDL et SystemC.

La commande de Modelsim pour compiler des fichiers SystemC est `sccom`. Elle utilise en interne une version de g++ fournie avec l'outil. Aussi, `sccom --link` doit être appelé après la compilation pour l'édition de liens.

`sccom` définit la macro `MTI_SYSTEMC` qui peut être testée pour écrire du code spécifique pour Modelsim.

---

4 Verilator

Convertir le Verilog en C++/SystemC

Utiliser un “convertisseur” qui converti le Verilog en C++/SystemC :

- `Verilator` : https ://www.veripool.org/wiki/verilator

Verilator est un outil qui analyse du code HDL (seulement Verilog ou SystemVerilog) et qui génère un AST (*Abstract Syntaxic Tree*).

À partir de cet AST il peut :

- générer un modèle SystemC ou C++ (qui utilise son propre modèle de simulation),
- mettre en œuvre un certain nombre d’optimisations :
  - simplification de logique combinatoire,
  - propagation de constantes,
  - faire les calculs sur des tailles adaptées aux processeurs…

Il a été développé pour générer des simulateurs rapides à partir de modèles RTL.

C’est un outil libre dont le code source est distribué sous les termes de la licence LGPLv3. Le code source est disponible ici et des versions binaires sont disponibles dans la majorité des distributions Linux.

Conversion automatique du RTL grâce à Verilator

Verilator permet donc de générer un modèle de simulation SystemC à partir d’un module Verilog ou SystemVerilog.

Le module généré peut alors être intégré dans une simulation SystemC standard.

Workflow `Verilator`

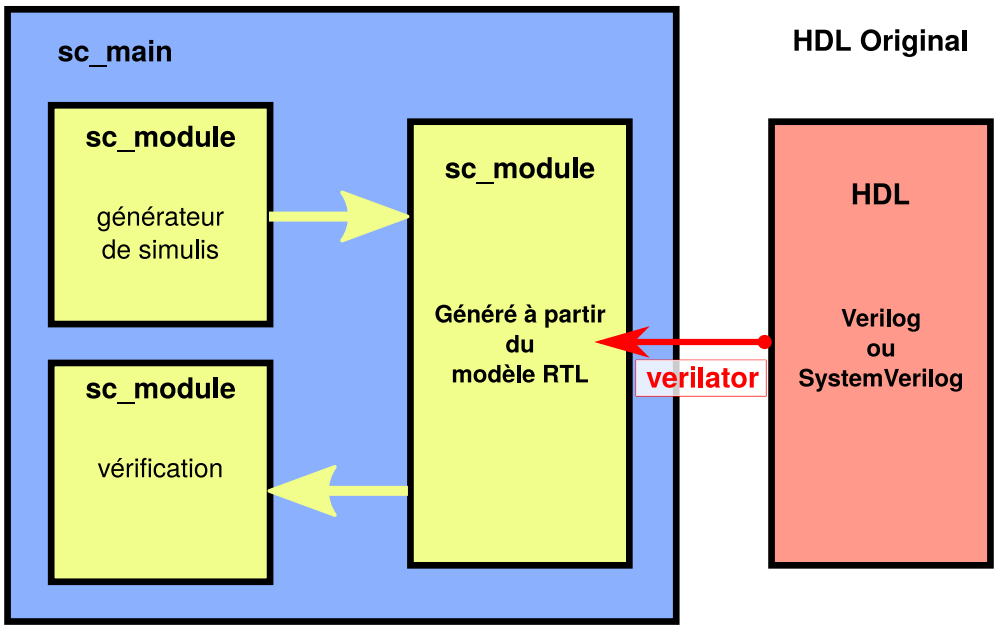

FIG. 4.1: Convertir le Verilog en SystemC

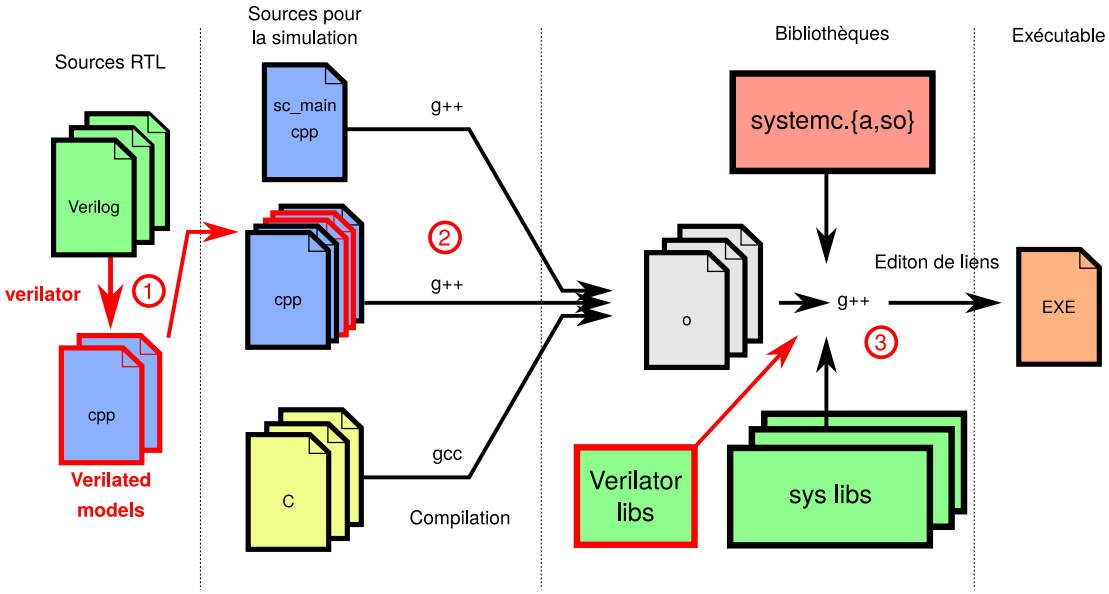

FIG. 4.2: Workflow

1. On utilise Verilator pour générer des modèles SystemC à partir des modules Verilog/SystemVerilog.
2. L'ensemble des sources est compilé en utilisant le compilateur C++ natif.
3. En plus de la bibliothèque SystemC, l'édition de liens doit utiliser certaines bibliothèques fournies avec l'outil.

L'outil sait aussi générer un *Makefile* pour automatiser la compilation.

Exemple (Conversion) :

Pour le module Verilog précédent (`counter`) on obtient le module SystemC `Vcounter` :

```
#ifndef _Vcounter_H_
#define _Vcounter_H_

#include "systemc.h"
#include "verilated_sc.h"
#include "verilated.h"

SC_MODULE(Vcounter) {
  public:
    sc_in<bool> clk;
    sc_in<bool> nrst;
    sc_in<bool> enable;
    sc_out<sc_uint<8> > Q;

    ....
}

#endif
```

Ce modèle peut être instancié directement dans un autre module SystemC ou directement dans le `sc_main`.

Pour obtenir ce module `verilator` est utilisé avec les options suivantes :

```
verilator -sc --pins-sc-uint counter.sv
```

La première option indique que l'on veut un modèle SystemC, la seconde qu'on veut des `sc_uint` pour les signaux de l'interface dont la largeur est supérieure à 1.

Si l'on souhaite spécialiser le code, la macro `VM_SC` peut être testée.

En réalité, le module généré est un peu plus complexe et contient des références à des objets internes de l'outil.

```
// Verilated -*- SystemC -*-
// DESCRIPTION: Verilator output: Primary design header
//
// This header should be included by all source files instantiating the design.
// The class here is then constructed to instantiate the design.
// See the Verilator manual for examples.

#ifndef _Vcounter_H_
#define _Vcounter_H_

#include "systemc.h"
#include "verilated_sc.h"
#include "verilated.h"

class Vcounter__Syms;

//----------

SC_MODULE(Vcounter) {
  public:

    // PORTS
    // The application code writes and reads these signals to
    // propagate new values into/out from the Verilated model.
    sc_in<bool> clk;
    sc_in<bool> nrst;
    sc_in<bool> enable;
    sc_out<sc_uint<8> > Q;

    // LOCAL SIGNALS
    // Internals; generally not touched by application code

    // LOCAL VARIABLES
    // Internals; generally not touched by application code
    VL_SIG8(__Vcellout__counter__Q,7,0);

    // INTERNAL VARIABLES
    // Internals; generally not touched by application code
    Vcounter__Syms* __VlSymsp;  // Symbol table

    // PARAMETERS
    // Parameters marked /*verilator public*/ for use by application code

    // CONSTRUCTORS
```

```
  private:
    VL_UNCOPYABLE(Vcounter);  ///< Copying not allowed
  public:
    SC_CTOR(Vcounter);
    virtual ~Vcounter();

    // API METHODS
  private:
    void eval();
  public:
    void final();

    // INTERNAL METHODS
  private:
    static void _eval_initial_loop(Vcounter__Syms* __restrict vlSymsp);
  public:
    void __Vconfigure(Vcounter__Syms* symsp, bool first);
  private:
    static QData _change_request(Vcounter__Syms* __restrict vlSymsp);
    void _ctor_var_reset();
  public:
    static void _eval(Vcounter__Syms* __restrict vlSymsp);
  private:
#ifdef VL_DEBUG
    void _eval_debug_assertions();
#endif // VL_DEBUG
  public:
    static void _eval_initial(Vcounter__Syms* __restrict vlSymsp);
    static void _eval_settle(Vcounter__Syms* __restrict vlSymsp);
    static void _settle__TOP__1(Vcounter__Syms* __restrict vlSymsp);
} VL_ATTR_ALIGNED(128);

#endif // guard
```

Par défaut, ce fichier d'en-tête, ainsi qu'un fichier C++, sont générés dans un sous-répertoire `obj_dir`. La méthode `eval` est l'unique méthode SystemC (`SC_METHOD`) dans laquelle se trouve tout le comportement du module RTL. L'observation des signaux interne n'est pas donc possible par défaut pour obtenir la meilleure performance.

On peut aussi demander à Verilator de générer un `Makefile` pour automatiser la compilation. Par exemple :

```
verilator -sc --pins-sc-uint -exe counter.sv sc_main.cpp -o ../sim.exe
```

Générera un Makefile (`Vcount.mk`) dans le répertoire `obj_dir` permettant de compiler un exécutable nommé `sim.exe` dans le répertoire courant.

Pour obtenir la documentation de l'outil et la liste des options :

```
verilator --help
```

---

5 Travail à faire :

Reprendre l'exemple du PGCD.

Voici le code SystemVerilog d'un module RTL calculant le PGCD de deux nombres codés sur 8 bits. Il possède la même interface et le même comportement que le module demandé dans le chapitre "Niveaux d'abstraction et raffinement".

```
module PGCD (
               input clk,
               input valid,
               input [7:0] a,
               input [7:0] b,
               output ready,
               output [7:0] pgcd
            );

logic [7:0] max, min;
wire [7:0] d = max-min;

always_ff @(posedge clk)
   if (valid)
   begin
      max <= (a>b)? a:b;
      min <= (a>b)? b:a;
   end
   else
   begin
      max <= (d>min)? d : min;
      min <= (d>min)? min : d;
   end

assign pgcd = max;
assign ready = (min==max);

endmodule
```

Travail à faire : mettre en place un environnement de simulation en SystemC en reprenant le `sc_main` existant, permettant de simuler ce module SystemVerilog en utilisant Modelsim et Verilator.

- Le `sc_main` doit instancier un module `testeur` qui génère des séquences de test et vérifie la validité du résultat renvoyé.
- Le code du `sc_main` doit être compatible avec les deux outils.
- **Bonus** écrire un Makefile qui automatise la compilation dans les deux cas.
