

Modélisation Transactionnelle des Systèmes sur Puces en SystemC Ensimag 3A — filière SLE Grenoble-INP Éléments de base

Matthieu Moy
(transparents originaux de Jérôme Cornet)

Matthieu.Moy@imag.fr

2012-2013



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 1 / 48 >

Planning approximatif des séances

- 1 Introduction : les systèmes sur puce
- 2 Introduction : modélisation au niveau transactionnel (TLM)
- 3 Introduction au C++
- 4 **Présentation de SystemC, éléments de base**
- 5 Communications haut-niveau en SystemC
- 6 Modélisation TLM en SystemC
- 7 TP1 : Première plateforme SystemC/TLM
- 8 Utilisations des plateformes TLM
- 9 TP2 (1/2) : Utilisation de modules existants (affichage)
- 10 TP2 (2/2) : Utilisation de modules existants (affichage)
- 11 Notions Avancé en SystemC/TLM
- 12 TP3 (1/3) : Intégration du logiciel embarqué
- 13 TP3 (2/3) : Intégration du logiciel embarqué
- 14 TP3 (3/3) : Intégration du logiciel embarqué
- 15 Intervenant extérieur
- 16 Perspectives et conclusion



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 2 / 48 >

Sommaire

- 1 Présentation
- 2 SystemC : Fondamentaux
- 3 SystemC : Modèle d'exécution, scheduler



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 3 / 48 >

Motivations (1/2)

- VHDL, Verilog standardisés (IEEE 1076-xxxx, IEEE 1364-xxxx)
- Langage de conception système :
 - ▶ Plusieurs niveaux d'abstraction (> RTL)
 - ▶ Intégration matériel et logiciel
 - ▶ Spécification exécutable
 - ▶ Simulation rapide



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 5 / 48 >

Motivations (2/2)

- Conception système : tout un monde de langages...
 - ▶ Handel C (Celoxica)
 - ▶ SystemVerilog (Accellera, Synopsys)
 - ▶ SpecC (UC Irvine)
 - ▶ Ptolemy (Berkeley)...
- Besoins du monde industriel
 - ▶ Langage (re)connu
 - ▶ Supporté par les vendeurs d'outils (**CAD vendors**)
 - ▶ Indépendant d'un vendeur ou d'une université en particulier



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 6 / 48 >

SystemC



- Proposition de Synopsys, Cadence et CoWare
- Langage... ou bibliothèque ?
- Concrètement
 - ▶ Ensemble de classes C++
 - ▶ Noyau de simulation (**scheduler**)
- Open source (licence type BSD)
- Standardisé
 - ▶ Accellera Systems Initiative (ASI), anciennement Open SystemC Initiative (OSCI)
 - ▶ IEEE 1666 (Décembre 2005, révisée en 2011)



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 7 / 48 >

SystemC : versions

- Petit historique
 - ▶ 2000 : SystemC 1.0 (RTL)
 - ▶ 2001 : SystemC 2.0 (Communications abstraites)
 - ▶ 2004 : Débuts de la bibliothèque **TLM OSCI**
 - ▶ Mars 2007 : **SystemC 2.2** (meilleur support TLM, etc.)
 - ▶ Juin 2008 : TLM 2.0
- 2011 : norme IEEE révisée
 - ▶ Contrôle des processus (suspend/resume, ...) pour modélisation des OS
 - ▶ Canaux primitifs « Thread-safe »
 - ▶ TLM 2.0 intégré au standard



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 8 / 48 >

SystemC : contenu

- Organisation

Methodology-Specific Libraries <i>Master/Slave library, etc.</i>	Layered Libraries <i>Verification library TLM library, etc.</i>
Primitive Channels <i>signal, fifo, mutex, semaphore, etc.</i>	
Structural elements <i>modules ports interfaces channels</i>	Data Types <i>4-valued logic Bits and Bit Vectors Arbitrary Precision Integers Fixed-point types</i>
Event-driven simulation <i>events processes</i>	
C++ Language Standard	



Matthieu Moy (Matthieu.Moy@imag.fr)

Modélisation TLM

2012-2013

< 9 / 48 >

Quelques types de base...

- Valeurs logiques simples
 - ▶ **bool** : type C++ natif, valeurs **true** et **false**
 - ▶ **sc_bit** : valeurs 0 et 1
 - ▶ **sc_logic** : quatre valeurs possibles
 - ★ '0', '1' : valeurs Booléennes false et true.
 - ★ 'X' : indéfini,
 - ★ 'Z' : haute impédance.
 - ★ ⇒ en général, pas utile en TLM!
- Vecteurs de valeurs logiques
 - ▶ **sc_bv<nbbits>** : vecteurs de **sc_bit**
 - ▶ **sc_lv<nbbits>** : vecteurs de **sc_logic**
 - ▶ **sc_int<nbbits>**, **sc_uint<nbbits>** : entiers
- Remarque : utilisation de la généricité de C++



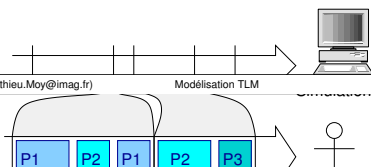
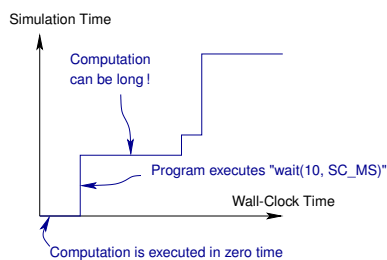
Représentation du temps

- Classe **sc_time** : couple (valeur, unité de temps)
- Unités
 - ▶ **SC_SEC** : seconde
 - ▶ **SC_MS** : milliseconde
 - ▶ **SC_US** : microseconde
 - ▶ **SC_NS** : nanoseconde
 - ▶ etc.

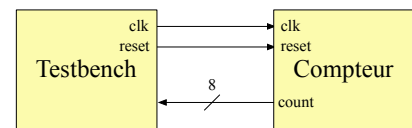


Temps simulé Vs « Wall-Clock Time »

- Temps simulé = temps que prendrait la puce pour faire la même chose
- Wall Clock Time = temps pris par la simulation.



Exemple de ce que l'on souhaite modéliser



Code associé

```
int sc_main(int, char**)
{
    Compteur      compteur("Compteur");
    Testbench     testbench("Testbench");
    sc_signal<bool> sclk, sreset;
    sc_signal<sc_uint<8> > scount;

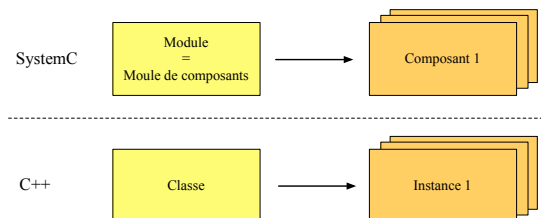
    testbench.clk.bind(sclk);
    testbench.reset.bind(sreset);
    testbench.count.bind(scount);

    compteur.clk.bind(sclk);
    compteur.reset.bind(sreset);
    compteur.count.bind(scount);

    sc_start(500, SC_NS); // penser a demarrer la simu!
    return 0;
}
```



Découpage en composants



Modules : déclaration (C++ « pur »)

```
struct Additionneur : public sc_module
{
    // attributs de la classe
    ...

    // constructeur
    ...
    Additionneur(sc_module_name nominstance);
    ...
};
```

- Classe mère commune pour tous les modules
 - ▶ Nom pour chaque composant
 - ▶ Rattachement au noyau de simulation



Modules : déclaration SystemC

- Utilisation de macros


```
SC_MODULE(Additionneur)
{
    // attributs de la classe
    ...

    // methodes de la classe
    SC_CTOR(Additionneur);
    ...
};
```
- **SC_MODULE**(user_module_name) : macro pour


```
struct user_module_name : ::sc_core::sc_module
```



Modules : instantiation

- Instantiation d'un module (objet C++)

```
#include "systemc.h"
#include "Additionneur.h"

int sc_main(int argc, char **argv)
{
    Additionneur add1("Composant1");

    ...

    return 0;
}
```

- On retrouve le nom pour chaque composant
- Autres paramètres de construction ?



Modules : implémentation

- Déclaration « manuelle » du constructeur (fichier .h)

```
SC_MODULE(ComposantP) {
    // cf transparent sur les processus
    // Les deux lignes suivantes remplacent SC_CTOR.
    SC_HAS_PROCESS(ComposantP);
    ComposantP(sc_module_name name, int parametre);
    ...
};
```

- Implémentation du constructeur (fichier .cpp)

```
#include "ComposantP.h"

ComposantP::ComposantP(sc_module_name name,
                      int parametre)
    : sc_module(name) {
    ...
}
```



Connexions simples (1/2)

- Ports de base : `sc_in<type>`, `sc_out<type>`, `sc_inout<type>`
- Connexions entre ports : `sc_signal<type>`
- Exemple :

```
SC_MODULE(Add8bits)
{
    // attributs de la classe
    sc_in<sc_uint<8>> a, b; // entrees additionneur
    sc_out<sc_uint<8>> c;    // sorties additionneur
    // attention aux bons espaces entre signes ">"

    // methodes de la classe
    SC_CTOR(Add8bits);
    ...
};
```



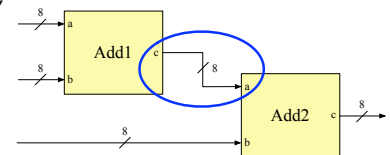
Connexions simples (2/2)

- Exemple de connections : additionneur 3 opérands

```
int sc_main(int argc, char **argv) {
    Add8bits add1("Add1"), add2("Add2");
    sc_signal<sc_uint<8>> s("s");

    add1.c.bind(s); // peut etre abrege add1.c(s)
    add2.a.bind(s);

    ...
    return 0;
}
```



Processus

- Modélisation de circuits électroniques : besoin de concurrence
- Deux formes de parallélisme :
 - ▶ Parallélisme de description : différents **processus** à l'intérieur des composants
⇒ on simule un système parallèle.
 - ▶ Parallélisme d'implémentation : exécution sur plusieurs processeurs (pas dans le cas de SystemC !)
⇒ on simule un système **en** parallèle (ou pas).
- Noyau de simulation SystemC
 - ▶ Choix du scheduling non préemptif
 - ▶ Les processus décident quand rendre la main
- Besoin de deux ingrédients :
 - ▶ Atomicité
 - ▶ « Rendre la main »



SC_METHOD : présentation

- Processus à exécution atomique (pas de wait)
- Création et exécution à la suite d'un **événement**
- Exemples d'événements
 - ▶ Changement de valeur d'un signal
 - ▶ Fronts montants, descendants
 - ▶ Événements utilisateurs (classe `sc_event`)
- Ensemble des événements déclenchant une SC_METHOD : **liste de sensibilité**



SC_METHOD : exemple combinatoire (1/2)

- Exemple additionneur Add8bits (déclaration complète) :

```
SC_MODULE(Add8bits)
{
    // attributs de la classe
    sc_in<sc_uint<8>> a, b; // entrees additionneur
    sc_out<sc_uint<8>> c;    // sorties additionneur

    // methodes de la classe
    SC_CTOR(Add8bits);

    void calcul(); // point d'entree de la SC_METHOD
    // pour l'instant, c'est une methode C++ normale
};
```



SC_METHOD : exemple combinatoire (2/2)

- Exemple additionneur Add8bits (implémentation) :

```
// fichier Add8bits.cpp
#include "Add8bits.h"

Add8bits::Add8bits(sc_module_name nom) : sc_module(nom)
{
    SC_METHOD(calcul); // calcul devient une SC_METHOD
    sensitive << a << b; // qui se "veille" quand
                        // a ou b change.

    // execution de calcul a chaque changement de a ou b
    void Add8bits::calcul()
    {
        c.write(a.read() + b.read());
        // peut s'ecrire c = a + b;
    }
}
```



SC_METHOD : exemple séquentiel (1/2)

- Sensibilité sur fronts : `monport.pos()` et `monport.neg()`
- Exemple : Compteur 8 bits

```
SC_MODULE(Compteur)
{
    // attributs de la classe
    sc_in<bool> clk;    // entree horloge
    sc_in<bool> reset;  // remise a zero

    sc_out<sc_uint<8> > count; // valeur

    // methodes de la classe
    SC_CTOR(Compteur);

    void calcul(); // point d'entree de la SC_METHOD
};
```



SC_METHOD : exemple séquentiel (2/2)

- Exemple : Compteur 8 bits (suite)

```
Compteur::Compteur(sc_module_name nom) : sc_module(nom)
{
    SC_METHOD(calcul);
    sensitive << clk.pos();
    sensitive << reset;
}

void Compteur::calcul()
{
    if (reset.read() == true)
        count.write(0);
    else if (clk.posedge())
    {
        count.write(count.read() + 1);
    }
}
```



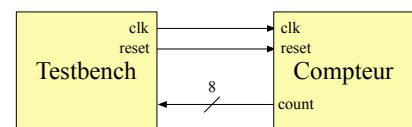
SC_THREAD : présentation

- Processus général, non atomique
- Par défaut : exécution au démarrage de la simulation
- « Rendre la main » :
 - ▶ Attente de temps : `wait(duree)`
 - ▶ Attente sur événement : `wait(eventement)`
- Exemple : module de test

```
Testbench::Testbench(sc_module_name nom)
    : sc_module(nom)
{
    SC_THREAD(genClk);
    SC_THREAD(genReset);
}
```



Schéma du module de test



SC_THREAD : attente sur du temps (1/2)

- Exemple : module de test (génération de reset)

```
void Testbench::genReset()
{
    reset.write(false);

    wait(2, SC_NS);

    reset.write(true);

    wait(5, SC_NS);

    reset.write(false);
}
```



SC_THREAD : attente sur du temps (2/2)

- Exemple : module de test (génération d'horloge)

```
void Testbench::genClk()
{
    clk.write(false);
    wait(10, SC_NS);

    while (true)
    {
        wait(3, SC_NS);

        if (clk.read() == false)
            clk.write(true);
        else
            clk.write(false);
    }
}
```



Instanciation complète

```
int sc_main(int, char**)
{
    Compteur compteur("Compteur");
    Testbench testbench("Testbench");
    sc_signal<bool> sclk, sreset;
    sc_signal<sc_uint<8> > scount;

    testbench.clk(sclk);
    testbench.reset(sreset);
    testbench.count(scount);

    compteur.clk(sclk);
    compteur.reset(sreset);
    compteur.count(scount);

    sc_start(500, SC_NS); // penser a demarrer la simu!
    return 0;
}
```



SC_THREAD : attente sur événement (1/3)

- Exemple précédent en utilisant des événements
- Définition du module :

```
SC_MODULE(Testbench)
{
    sc_out<bool> clk;
    sc_out<bool> reset;

    sc_in<sc_uint<8> > count;

    // implementation du constructeur inchangée
    SC_CTOR(Testbench);

    sc_event reset_finished;
};
```



SC_THREAD : attente sur événement (2/3)

- Exemple précédent en utilisant des événements
- Génération du reset :

```
void Testbench::genReset()
{
    reset.write(false);

    wait(2, SC_NS);

    reset.write(true);

    wait(5, SC_NS);

    reset.write(false);
    // notification de l'événement
    reset_finished.notify();
}
```



SC_THREAD : attente sur événement (3/3)

- Exemple précédent en utilisant des événements
- Définition du module :

```
void Testbench::genClk()
{
    clk.write(false);
    wait(reset_finished); // attente d'événement

    while (true)
    {
        wait(3, SC_NS);

        if (clk.read() == false)
            clk.write(true);
        else
            clk.write(false);
    }
}
```



SC_THREAD : variante

- Mise de l'événement en liste de sensibilité
- Exemple :

```
// constructeur
void Testbench::Testbench(sc_module_name name)
    : sc_module(name)
{
    SC_THREAD(genClk);
    sensitive << reset_finished;
}

void Testbench::genClk()
{
    clk.write(false);
    wait(); // attente d'événement

    while (true)
    ...
}
```



Exercice

Question



Comment faire un additionner générique n bits ?



Parallelism in SystemC

Discrete-event simulation

- One global timescale,
- SystemC contains a scheduler,
- Scheduler manages a list of processes and an agenda.

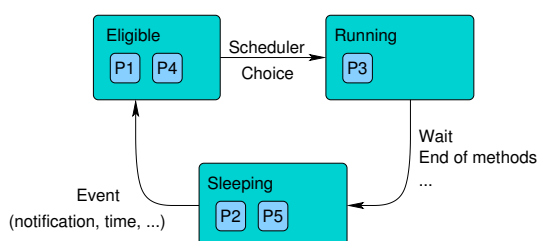


SystemC Scheduler : The Agenda

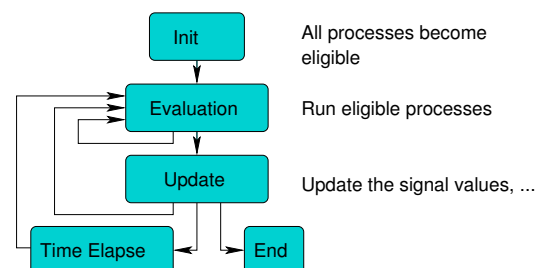
- List of "things to do later",
- Executing process usually program events in the future,
- When nothing more is to be done in the present, go to the next thing to do in the future.



SystemC Scheduler : List of Processes

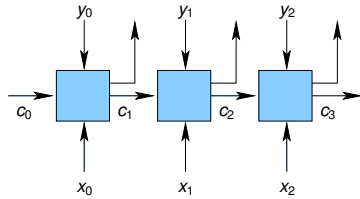


SystemC Scheduler : Scheduling Algorithm



Why an Update Phase ?

- Example : n bit adder in RTL :



- Each cell executes :
 - ▶ $o_i = x_i \text{ xor } y_i$
 - ▶ $c_{i+1} = x_i \cdot y_i$
- Data-dependency !



Why an Update Phase ?

- Possible approaches :
 - ▶ **Physical circuit** : the carry propagates, the last signals might oscillate briefly and stabilize.
 - ▶ **Synchronous languages** : static data-dependency
 - ★ Needs more work in the compiler
 - ★ Forbids some constructs that would still have worked (if statements, separate compilation problem, ...)
 - ▶ **SystemC/VHDL/... : δ -cycles**
 - ★ Evaluate all the processes in any order,
 - ★ Re-run the processes whose input changed until stabilization.



δ -Cycle and Update Phase

- Most actions take effect at the end of the δ -cycle :
 - ▶ `port.write(value);`,
 - ▶ `event.notify(SC_ZERO_TIME);`,
 - ▶ `wait(SC_ZERO_TIME);`,
 - ▶ ...
- Order of execution within the δ -cycle *should* not matter,

Question



what does a self-loop on a not-gate do ?



δ -cycles and TLM

- δ -cycles mostly come from the RTL world
- Can be used to model complex zero-time behavior
- Usually bad practice \Rightarrow don't rely on them



Conclusion sur cette partie

- Mécanismes de modélisation RTL
- Partie non RTL : testbench... comme en VHDL !
- Éléments de base aussi utilisés par la suite
- Synthétisabilité
 - ▶ SystemC Synthesizable Subset (Draft 1.3) (e.g. `SC_METHOD` sans allocation de pointeurs)
 - ▶ Utilisation des types appropriés (bool, `sc_logic`...)
 - ▶ Quelques outils commerciaux :
 - ★ CoCentric SystemC Compiler (Synopsys) (abandonné, puis ressuscité)
 - ★ Nepsys (Prosilog)
 - ★ Synthesizer (Forte Design Systems)
 - ★ PICO (code C)
 - ▶ Utilité ?

