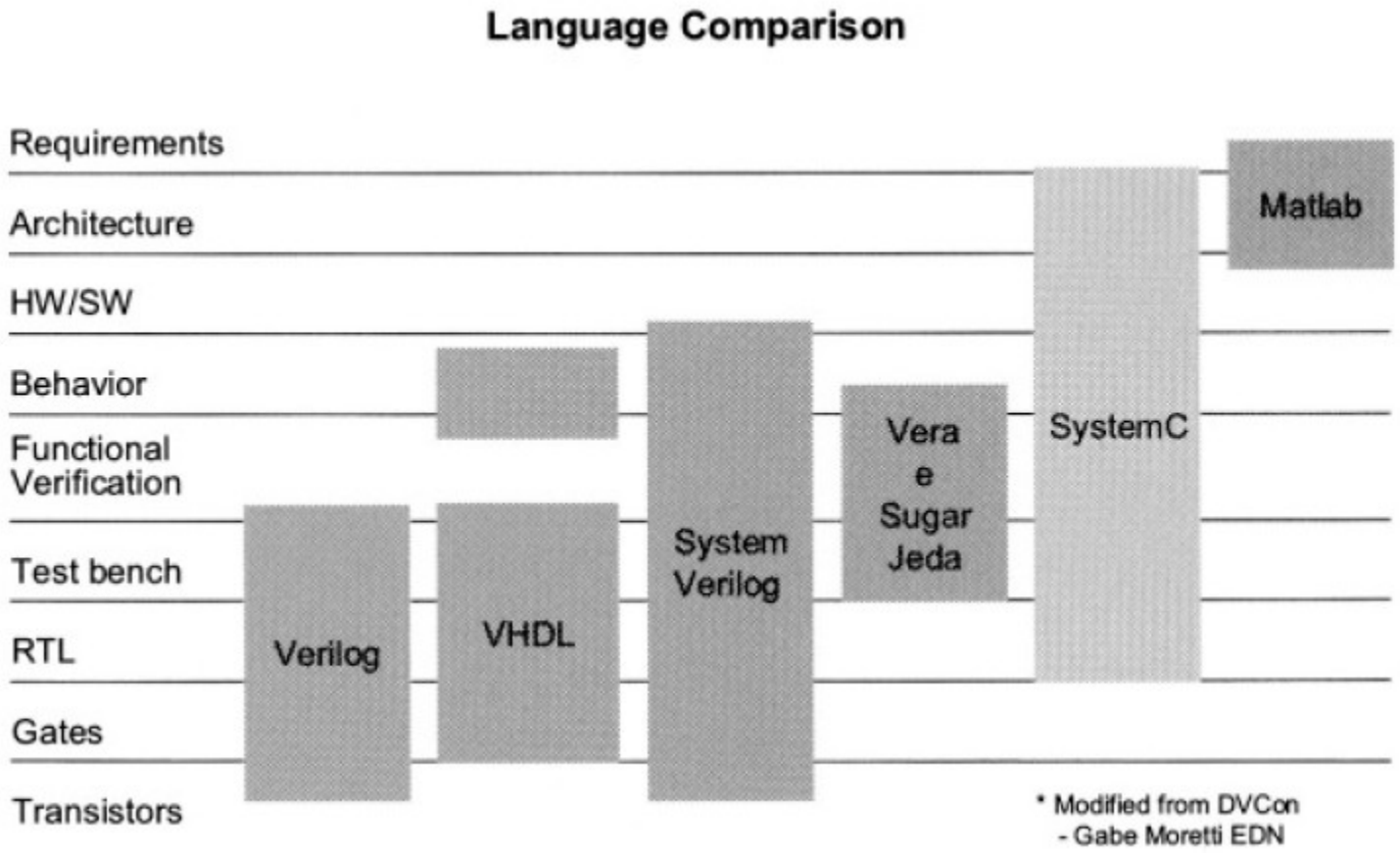
Wprowadzenie do SystemC

SystemC - porównanie



Moduły

Deklaracja:

```
  SC_MODULE(transmit) {
```

Porty modułów:

```
SC_MODULE(fifo) {
  sc_in<bool > load;
  sc_in<bool > read;
  sc_inout<int > data;
  sc_out<bool > full;
  sc_out<bool > empty;
//rest of module not shown
}
```

Moduły - konstruktor

```
SC_CTOR(module_name)
: Initialization // OPTIONAL
{
  Subdesign_Allocation
  Subdesign_Connectivity
  Process_Registration
  Miscellaneous_Setup
}
```

Zadania konstruktorów:

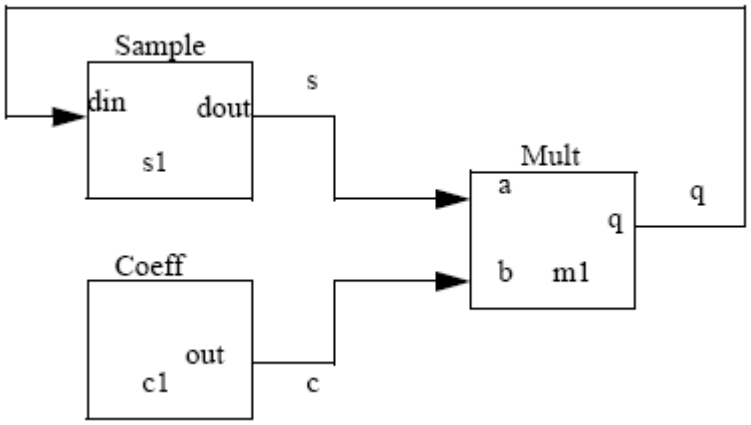
- Alokacja i inicjalizacja modułów podrzędnych (w hierarchii).
- Tworzenie połączeń z portami modułów podrzędnych.
- Rejestracja procesów (w jądrze SystemC w celu modelowania współbieżności).
- Sterowanie czułością procesów.
- Inne zadania – zdefiniowane przez użytkownika.

Moduły - połączenia pozycyjne

```
// filter.h
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"
SC_MODULE(filter) {
  sample *s1;
  coeff *c1;
  mult *m1;
  sc_signal<sc_uint<32> > q, s, c;
  SC_CTOR(filter) {
    s1 = new sample ("s1");
    (*s1)(q,s);
    c1 = new coeff ("c1");
    (*c1)(c);
    m1 = new mult ("m1");
    (*m1)(s,c,q);
  }
}
```

Tworzenie nowej instancji modułu „sample” o nazwie s1.

Podłączenie lokalnych sygnałów q,s odpowiednio do portów din, dout modułu s1.



Moduły - połączenia nazywane

Przykład 1:

```
SC_MODULE(filter) {
  sample *s1;
  coeff *c1;
  mult *m1;
  sc_signal<sc_uint<32> > q, s, c;
  SC_CTOR(filter) {
    s1 = new sample ("s1");
    s1->din(q);
    s1->dout(s);
    c1 = new coeff ("c1");
    c1->out(c);
    m1 = new mult ("m1");
    m1->a(s);
    m1->b(c);
    m1->q(q);
  }
}
```

Przykład 2:

```
int sc_main(int argc, char* argv[])
{
  sc_signal<sc_uint<8> > dout1, dout2, dcntr;
  sc_signal<sc_logic > rst;

  counter_tb CNTR_TB("counter_tb"); // Instantiate
    Test Bench
  CNTR_TB.cnt(dcntr);

  ror DUT("ror"); // Instantiate Device Under Test
  DUT.din(dcntr);
  DUT.dout1(dout1);
  DUT.dout2(dout2);
  DUT.reset(rst);

  // Run simulation:
  rst=SC_LOGIC_1;
  sc_start(15, SC_NS);
  rst=SC_LOGIC_0;
  sc_start();
```

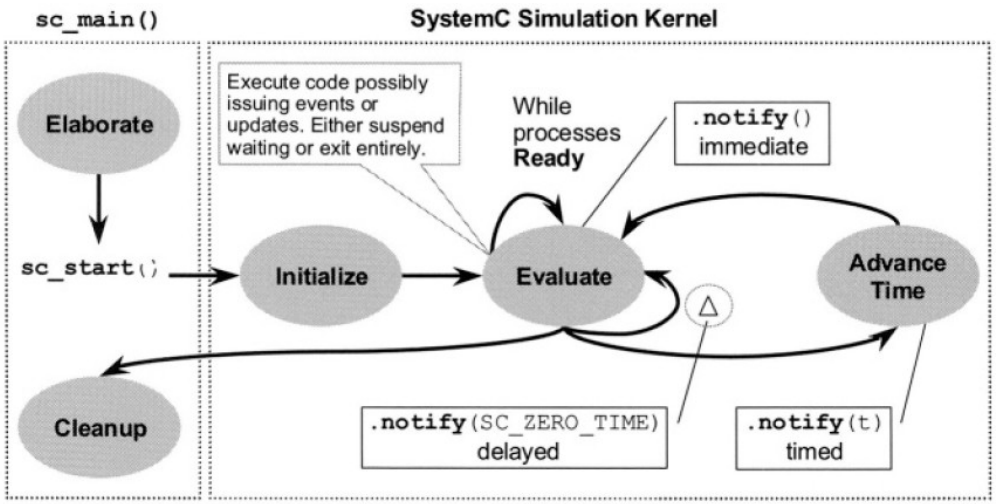
Moduły – funkcja sc_main

SystemC posiada własną wersję funkcji main() i z niej uruchamia funkcję sc_main zdefiniowaną przez użytkownika (kopiując parametry wywołania):

```
int sc_main(int argc, char* argv[]) {
  // ELABORATION
  sc_start();  // SIMULATION
  // POST-PROCESSING
  return EXIT_CODE; // Zero indicates success
}
```

W fazie elaboracji tworzone są instancje obiektów takich jak jednostki projektowe, zegary, kanały. Tworzona jest hierarchia połączeń pomiędzy modułami.
Oprócz tego rejestrowane są procesy.
Moduły są zazwyczaj włączane do projektu w plikach nagłówkowych *.h.

Jądro symulatora



Etapy działania symulatora:

- **Elaborate**

sc_start(1):

- **Initialize**
- **Evaluate**
  - Advance time
    - Delta cycles
- **Cleanup**

Procesy

Procesy są głównymi jednostkami wykonawczymi w SystemC. Są one uruchamiane w celu emulacji zachowania urządzenia lub systemu. Są trzy rodzaje procesów:

- **Methods**
- **Threads**
- **Clocked Threads**

Procesy c.d.

- Procesy mogą zachowywać się jak funkcje (wykonanie i powrót).
- Procesy mogą także być uruchamiane na początku symulacji i pracować do końca symulacji bądź być przerywane przez oczekiwanie na warunek (np. zbocze zegara).
- Procesy nie są hierarchiczne – tj. nie można uruchamiać jednego procesu bezpośrednio z innego (można to robić pośrednio - poprzez generowanie sygnałów wyzwalających).
- Procesy posiadają listę czułości (listę sygnałów których zmiany powodują uruchomienie procesu).
- Zmiana wartości sygnału nazywa się zdarzeniem na sygnale (event on signal).

SC_METHOD

- Proces SC_METHOD jest uruchamiany tylko wtedy, gdy nastąpi jakakolwiek zmiana sygnałów na liście czułości procesu (event).
- Następnie proces wykonuje swoje zadania (zmienia stany sygnałów) i w skończonym (i najlepiej krótkim) czasie kończy działanie przekazując sterowanie do jądra SystemC (proces czeka na następny event).
- Instrukcja wait() wewnątrz procesu SC_METHOD jest niedozwolona.

Przykład (przerzutnik D):

```
SC_MODULE(dff) {
    sc_in<bool > clock;
    sc_in<bool > din;
    sc_out<bool > dout;

    void proc1() {
        dout.write(din.read());
    }

    SC_CTOR(dff) {
        SC_METHOD(proc1);
           sensitive << clock.pos();
    }
};
```

SC_METHOD – c.d.

Przykład (przerzutnik D z asynchronicznym resetem):

```
SC_MODULE(dffr) {
    sc_in<bool > clock;
    sc_in<bool > din;
    sc_in<bool > reset;
    sc_out<bool > dout;

    void proc1() {
    if (reset.read())
        dout.write(0);
    else
        if (clock.event() && clock.read())
            dout.write(din.read());
    }

    SC_CTOR(dffr) {
        SC_METHOD(proc1);
           sensitive << clock.pos() << reset;
    }
};
```

SC_METHOD – czułość dynamiczna

W procesach można modyfikować warunki wyzwalania w sposób dynamiczny za pomocą instrukcji next_trigger. Przykłady:

```
next_trigger(time);
next_trigger(event);
next_trigger(event1 | event2);   // dowolne ze zdarzeń
next_trigger(event1 & event2);   // wszystkie zdarzenia
next_trigger(timeout, event);   // wersje z timeoutem
next_trigger(timeout, event1 | event2);
next_trigger(timeout, event1 & event2);
next_trigger();  // powrót do statycznych ustawień czułości
```

Wszystkie procesy są wyzwalane w chwili t=0.
Aby tego uniknąć należy użyć funkcji dont_initialize():

```
...
SC_METHOD(test);
sensitive(fill_request);
dont_initialize();
...
```

SC_THREAD

- Proces SC_THREAD jest uruchamiany jednorazowo przez symulator.
- Następnie proces wykonuje swoje zadania, a sterowanie do symulatora może oddać albo poprzez return (ostateczne zakończenie procesu) albo poprzez wykonanie instrukcji wait (czasem pośrednio – np. poprzez blokujący read lub write).
- Wait powoduje przejście procesu do stanu zawieszenia. W zależności od wersji instrukcji wait wznowienie procesu może nastąpić pod różnymi warunkami:

```
wait(time);
wait(event);
wait(event1 | event2);
wait(event1 & event2);
wait(timeout, event);
wait(timeout, event1 | event2);
wait(timeout, event1 & event2);
wait();
```

Sposób wykrywania zakończenia instrukcji wait poprzez timeout:

```
sc_event ok_event, error_event;
wait(t_MAX_DELAY, ok_event | error_event);
if (timed_out()) break; // wystąpił time out
```

SC_THREAD - zdarzenia

Sposób generowania zdarzeń (event):

```
event_name.notify(); // immediate notification
event_name.notify(SC_ZERO_TIME) ; // next delta-cycle notification
event_name.notify(time); // timed notification
```

Przykład modułu z procesem SC_THREAD:

```
SC_MODULE(simple_example) {
  SC_CTOR(simple_example) {
    SC_THREAD(my_process); }
  void my_process(void); };
```

Przykład uruchomienia symulatora:

```
int sc_main(int argc, char* argv[]) { // args unused
simple_example my_instance("my_instance");
sc_start(10,SC_SEC);
// sc_start(); // forever
return 0;   // simulation return code
}
```

Typ sc_string (literały)

```
sc_string name("0base[sign]number[e[+|-]exp]");
```

prefix	znaczenie	Sc_numrep
0d	Decimal	SC_DEC
0b	Binary	SC_BIN
0bus	Binary unsigned	SC_BIN_US
0bsm	Binary signed magnitude	SC_BIN_SM
0o	Octal	SC_OCT
0ous	Octal unsigned	SC_OCT_US
0osm	Octal signed magnitude	SC_OCT_SM
0x	Hex	SC_HEX
0xus	Hex unsigned	SC_HEX_US
0xsm	Hex signed magnitude	SC_HEX_SM
0csd	Canonical signed digit	SC_CSD

Przykłady:

```
sc_string a ("0d13"); // decimal 13
a = sc_string ("0b101110") ; // binary of decimal 44
```

Typy w SystemC

- Zalecane jest wykorzystywanie typów języka C++

- Typ odpowiadający typowi integer z języka VHDL (potrzebny np. do syntezy):
  sc_int<LENGTH> nazwa...;
  sc_uint<LENGTH> nazwa...;
  sc_int – integer ze znakiem o długości LENGTH bitów,
  sc_uint - integer bez znaku o długości LENGTH bitów.

- Typ integer o większej szerokości niż 64 bity:
  sc_bigint<BITWIDTH> nazwa...;
  sc_biguint<BITWIDTH> nazwa...;

- Typ bitowy (i wektorowy typ bitowy):
  sc_bit nazwa...;
  sc_bv<BITWIDTH> nazwa...;

- Typ logiczny (i wektorowy typ logiczny) – posiada 4 możliwe wartości (podobny do std_logic):
  sc_logic nazwa[,nazwa]...;
  sc_lv<BITWIDTH> nazwa[,nazwa]...;
  możliwe wartości: **SC_LOGIC_1,SC_LOGIC_0, SC_LOGIC_X, SC_LOGIC_Z**
  zapis w łańcuchach: **sc_lv<8> dat_x ("ZZ01XZ1Z");**

Typy w SystemC – typ stałoprzecinkowy

Należy zdefiniować przed #include <systemc.h>:

```
#define SC_INCLUDE_FX
```

```
sc_fixed<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME...;
sc_ufixed<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME...;
sc_fixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME...;
sc_ufixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME...;
sc_fix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
sc_ufix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
sc_fixed_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
sc_ufixed_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
WL – Word length (długość słowa)
IWL – Integer Word length (długość części całkowitej)
QUANT – Tryb kwantyzacji (default: SC_TRN)
OVFLW – Tryb obsługi przepełnienia (default: SC_WRAP)
NBITS – liczba bitów nasycenia
```

Nazwa	Tryb obsługi przepełnienia
SC_SAT	Nasycenie do +/- max.
SC_SAT_ZERO	Zerowanie w nasyceniu
SC_SAT_SYM	Symetryczne nasycenie
SC_WRAP	Wraparound (default)
SC_WRAP_SYM	Symetryczy wraparound

Nazwa	Tryb kwantyzacji
SC_RND	Zaokrąglenie (do +∞ przy równym dystansie)
SC_RND_ZERO	Zaokrąglenie (do 0 przy równym dystansie)
SC_RND_MIN_INF	Zaokrąglenie (do -∞ przy równym dystansie)
SC_RND_INF	Zaokrąglenie (do +/-∞ zależnie od znaku liczby przy równym dystansie)
SC_RND_CONV	Zbieżne zaokrąglenie
SC_TRN	Obcięcie (default)
SC_TRN_ZERO	Obcięcie do zera

Typy w SystemC – opis czasu

Możliwe jednostki czasu:

```
SC_SEC, SC_MS, SC_US (mikrosekundy),
SC_NS, SC_PS, SC_FS
```

Typ sc_time:

```
sc_time name...; // bez inicjalizacji
sc_time name (wartość, jednostka)...;  // z inicjalizacją
```

Przykłady użycia:

```
sc_time t_PERIOD(20, SC_NS) ;
sc_time t_TIMEOUT(10, SC_MS) ;
sc_time t_X, t_CUR, t_LAST;
t_X = (t_CUR-t_LAST) ;
if (t_X > t_MAX) { error ("Timing error"); }
```

Odczyt czasu symulacji:

```
cout << sc_time_stamp() << endl;
```