Introducere în modelare și simulare cu SystemC

Pal-Ștefan MURVAY

2018

Cuprins

1 Introducere

SystemC este un limbaj care permite modelarea şi simularea componentelor unui sistem. În fapt SystemC se prezintă sub forma unei librării C++ ce pune la dispoziţie, sub formă de clase, funcţii şi tipuri de date, toate mecanismele necesare modelării de componente hardware şi a interacţiunilor dintre acestea. Din această perspectivă SystemC este comparabil cu alte libaje descriptive precum VHDL sau Verilog. Implementarea sub forma unei librării C++ face posibilă modelarea componentelor hardware dintr-un sistem împreună cu cele software oferind un bun suport pentru aplicarea codesignului hardware-software.

Lucrarea de faţă introduce elementele de bază în utilizarea SystemC. Pentru informaţii detaliate despre funcţionalităţile puse la dispoziţie recomandăm consultarea documentaţiei ce însoţesc versiunile librăriei SystemC: specificaţia funcţională [1], manualul de referinţă al limbajului (disponibil până la versiunea 2.1 a librăriei) [2] şi ghidul utilizatorului [3]. Există şi o serie de cărţi care tratează subiectul însoţind descrierea SystemC cu exemple relevante. Un bun exemplu este cartea scrisă de Donovan şi Black [4].

2 Instalare SystemC

Înainte de a începe modelarea sistemelor cu SystemC trebuie să ne asigurăm că librăria este disponibilă pentru mediul de dezvoltare folosit. Librăria SystemC poate să fie folosită atât pe sisteme de operare Windows pentru a dezvolta modele folosind mediul Visual Studio cât şi pe sisteme Linux. În continuare sunt prezentaţi paşii necesari instalării librăriei SystemC (versiunea 2.3.2) pe un sistem de operare Linux. În particular, pentru comenzile ce necesită drepturi de administrator s-a folosit apelarea prin comanda `sudo` caracteristică distribuţiilor de Ubuntu.

1. Obţineţi sursele librăriei SystemC disponibile online la adresa http://www.accellera.org/downloads/standards/systemc şi asiguraţi-vă că pachetul build-essential este instalat pentru că va fi necesar la compilarea librăriei.
2. Deschideţi terminalul şi faceţi ca directorul curent să fie folderul rădăcină al librăriei SystemC 2.3.2

   ```
   $ cd systemc-2.3.2
   ```

3. Creaţi directorul în care se va instala librăria

   ```
   $ sudo mkdir /usr/local/systemc-2.3.2
   ```

4. Creaţi un director numit objdir in directorul curent şi alegeţi-l ca director curent

   ```
   $ mkdir objdir
   ```

```
$ cd objdir
```

5. Efectuaţi configurarea, compilarea şi instalarea pachetului

```
$ sudo ../configure --prefix=/usr/local/systemc-2.3.2
$ sudo make
$ sudo make install
```

6. Pentru a ne asigura că librăria SystemC este localizată corect la compilare trebuie să salvăm calea către aceasta într-o variabilă de sistem

```
$ export SYSTEMC_HOME=/usr/local/systemc-2.3.2/
```

7. Pentru ca această variabilă de sistem să fie disponibilă şi după repornirea sistemului de operare deschideţi fişierul /etc/environment şi adăugaţi în el noua înregistrare

```
$ sudo gedit /etc/environment
```

Adăugaţi următoarea linie în fişierul deschis după care salvaţi şi închideţi editorul: `SYSTEMC_HOME="/usr/local/systemc-2.3.2/"`

3 Primul program în SystemC

Primul program scris la învăţarea oricărui limbaj de programare este de obicei programul "Hello, World!". Păstrând aceeaşi abordare, vom începe şi în cazul SystemC cu un program care va scrie "Hello, World!". Acest program este redat în Figura 1.

```
#include "systemc.h"

SC_MODULE (hello_world) {
  SC_CTOR (hello_world) {
  }
  void write_hello() {
    cout << "Hello, World!\n";
  }
};

int sc_main(int argc, char* argv[])
{
  //Instantiate module
  hello_world hello("Hello");
  //Write "Hello, World!"
  hello.write_hello();

  return(0);
}
```

Figura 1. hello_world.cpp

Pentru a compila acest program, utilizaţi terminalul pentru a naviga în directorul în care se află codul sursă şi executaţi următoarea comanda:

```
g++ -I. -I$SYSTEMC_HOME/include -L. -L$SYSTEMC_HOME/lib-linux -Wl,-rpath=$SYSTEMC_HOME/lib-linux -o sim hello_world.cpp -lsystemc -lm
```

şi apoi executaţi programul rezultat:

```
./sim
```

Pentru sistemele de operare pe 64 de biţi rereferinţa *SYSTEMC_HOME/lib-linux* din comanda de compilare se înlocuieşte cu *SYSTEMC_HOME/lib-linux64*. Dacă totul decurge bine ar trebui ca în urma rulării să vedeţi afişată în consolă o secvenţă similară celei din Figura 2.

Figura 2. Secvenţa afişată în urma rulării programului "Hello, World!"

În acest prim exemplu se pot observa câteva din elementele pe care le regăsim în toate programele SystemC. Acestea sunt uşor de identificat pentru că resursele diponibile în librăria SystemC (tipuri de date, funcţii, macrouri, etc.) au, de regulă, denumirea prefixată cu "`SC_`" sau "`sc_`".

Orice program ce foloseşte funcţionalităţi SystemC va include headerul corespunzător `systemc.h`. În continuare se foloseşte macroul `SC_MODULE` pentru a defini un modul – elementul de bază pentru construcţia unui model în SystemC. Similar unei clase în C++ modulul `hello_world` conţine un constructor `SC_CTOR` şi o funcţie (metodă) folosită pentru a tipării textul de salut. În cazul SystemC funcţia principală, echivalentă conceptual cu binecunoscuta funcţie `main`, este denumită `sc_main`. Aici se instanţiază modulul `hello_world` şi se apelează funcţia `write_hello`. Toate aceste elemente vor fi detaliate în secţiunile următoare.

4 Tipuri de date

Fiind construit ca o librărie C++, SystemC suportă în mod nativ toate tipurile de date disponibile în C/C++: întregi cu sau fără semn, reale şi bool. În cele mai multe cazuri, utilizarea tipurilor de date native C++ sunt suficiente pentru realizarea modelelor SystemC, acestea fiind şi cele mai eficiente atât în ceea ce priveşte consumul de memorie cât şi în privinţa vitezei de execuţie a simulării. Dacă însă se doreşte sinteza logică a modelului descris în SystemC este obligatorie utilizarea tipurilor de date dedicate puse la dispoziţie de librărie.

5 Module

Modulul este elementul constructiv de bază în SystemC ce reprezintă starea, comportamentul şi interfeţele componentei modelate. Conceptul de modul în SystemC este similar unui modul definit în Verilog sau unui element de tip Entity/Architecture în VHDL. Din perspectiva libajului C++, un modul SystemC nu este altceva decât o clasă derivată din clasa de bază a tuturor modulelor `sc_module`. În mod uzual declararea unui modul se face folosind macroul SC_MODULE ca în Figura 3, unde SC_MODULE este doar un macro a cărui definiţie este, în esenţă, echivalentă cu următoarea definiţie:

`#define SC_MODULE(`*`modul`*`) class `*`modul`*`: public sc_module`.

Dacă nu se doreşte utilizarea acestui macro, se pot declara module utilizând sintaxa standard C++ pentru definirea claselor ţinând cont de faptul că un modul trebuie să fie derivat din clasa de bază `sc_module` pentru a putea beneficia de funcţionalptăţile specifice SystemC. Fiind în esenţă o clasa, declararea unui modul trebuie întotdeauna urmată de caracterul '*;*'.

Un modul este constituit dintr-o serie de elemente specifice SystemC precum: constructor, porturi, structuri interne de date, instanţe de sub-module, procese şi canale de comunicare între procese.

```
SC_MODULE(nume_modul) {
      SC_CTOR(nume_modul) {
            // Corpul constructorului
      }
};
```

Figura 3. Structura minimală a unui modul SystemC

5.1 Constructorul

Componenta `SC_CTOR` din declaraţia modulului prezentat în figura anterioară este constructorul clasei a cărui prezenţă este obligatorie în orice modul. Constructorul declarat prin macroul `SC_CTOR` are ca singur argument un string reprezentând numele asociat instanţei modulului. Fiecărei instanţe ale unui modul ise asociază un nume pentru a ajuta în identificarea operaţiunilor realizate de fiecare instanţă în timpul execuţiei simulării. Numele atribuite instanţelor trebuie să fie unice. Pe lângă funcţionalităţile clasice pe care le are un constructor în C++ constructorul unui modul este responsabil de alocarea, iniţializarea şi conectarea sub-componentelor şi înregistrarea proceselor concurente.

Există cazuri în care constructorul standard `SC_CTOR` nu poate să fie utilizat. Un astfel de caz ar fi situaţia în care avem nevoie de constructori cu parametrii suplimentari. Întâlnim aceeaşi situaţie şi atunci când se doreşte separarea declaraţiei unui modul de definiţia acestuia plasând definiţia într-un fişier `.cpp`

separat. În aceste cazuri, în declaraţia modulului se invocă macroul SC_HAS_PROCESS urmat de constructorul scris folosind sintaxţa convenţională C++. Astfel, dacă definim constructorul odată cu declararea lui vom folosi o abordare similară celei din Figura 4, iar în cazul separării declaraţiei de implementare abortarea din Figura 5.

```
SC_MODULE(nume_modul) {
      SC_HAS_PROCESS(nume_modul);
      nume_modul(sc_module_name nume_instanta [, parametrii_aditionali])
      : sc_module(nume_instanta)
      {
            //Corp constructor
      }
};
```

Figura 4. Utilizarea SC_HAS_PROCESS la definirea constructorului în fişierul header

```
SC_MODULE(nume_modul) {
      SC_HAS_PROCESS(nume_modul);
      nume_modul (sc_module_name nume_instanta [,parametrii_aditionali]);
};
```

```
nume_modul::nume_modul(
      sc_module_name nume_instanta [, parametrii_aditionali])
      : sc_module(nume_instanta)
      {
            //Corp constructor
      }
```

Figura 5. Utilizarea SC_HAS_PROCESS la definirea în fişier .h şi implementarea separată în fişier .cpp

5.2 Procese

Procesul reprezintă unitatea elementară de execuţie în SystemC care oferă suportul pentru simularea execuţiei concurente. Procesul se declară ca o metodă membră a clasei, metodă ce nu are parametri şi nu returnează o valoare:

`void nume_proces(void);`.

Un proces nu poate să fie apelat direct de către un alt proces. Doar kernelul de simulare SystemC poate apela un proces. Există două tipuri de procese des utilizate în modelarea cu SystemC: SC_THREAD şi SC_METHOD. Atribuirea comportamentului de thread sau metodă se realizează prin înregistrarea tipului unui proces în interiorul constructorului folosind un macroul corespunzător. Astfel pentru înregistrea unui proces ca metodă se va adăuga în constructor secvenţa de cod: `SC_THREAD(nume_proces);`. În mod similar, pentru înregistrarea procesului ca metodă se va scrie `SC_METHOD(nume_proces);`. Aceste secvenţe de cod transmit kernelului de simulare SystemC care sunt procesele şi ce comportament să le atribuie. O funcţie care nu a fost înregistrată ca proces în constructor va avea, în timpul simulării, comportamentul standard conform limbajului C++.

Procesele de tip SC_THREAD se comportă ca un thread software, permiţând execuţia în paralel a mai multor procese de acest tip. Execuţia unui SC_THREAD porneşte la începutul simulării şi se termină odată cu terminarea execuţiei codului aferent. Un SC_THREAD care şi-a terminat execuţia nu poate să fie repornit. Există metode prin care execuţia unui SC_THREAD poate să nu înceapă odată cu startul simulării ci este determinată de apariţia unor evenimente. Chiar şi în acest caz apelul este permis o singură dată pe durata unei simulări. În mod uzual implementările de procese de tip thread conţin o buclă infinită ce execută o secvenţă de cod după care îşi suspenda execuţia pentru o perioadă de timp (printr-un apel al funcţiei metoda `wait`) permiţând continuarea execuţiei altor procese.

Procesele de tip SC_METHOD, după cum sugerează şi numele macroului, au un comportament similar unor metode clasice C++. Ele sunt apelate de către kernelul de simulare şi sunt executate în întregime fără să fie permisă suspendarea lor prin apelarea metodei `wait` (regulă valabilă atât pentru apelurile directe cât şi pentru cele indirecte). Metodele pot fi executate ori de câte ori este necesar pe durata simulării kernelul apelându-le atunci când au loc evenimentele aflate în lista de sensitivitate a procesului. Despre evenimente şi liste de sensitivitate se discută în detaliu în secţiunea 8. La fel ca în cazul altori tipuri de procese, la începutul simulării kernelul va apela implicit şi metodele.

Sunt unele cazuri în care nu se doreşte apelarea tuturor proceselor la începutul simulării. Acest comportament poate fi împiedicat pentru procesele ce nu trebuie rulate prin adăugarea în constructor a unui apel către funcţia `dont_initialize()` imediat după înregistrarea procesului ca metodă sau thread. În astfel de cazuri trebuie asigurat apelul procesului de către kernelul de simulare prin alte mecanisme.

Pentru a uşura înţelegerea codului se foloseşte drept convenţie de numire a proceselor adăugatea unui sufix care să indice tipul procesului, de exemplu `proces1_thread` sau `proces2_method`.

6 Realizarea unui Test Bench

Punctul de pornire al unei simulări SystemC este funcţia `sc_main`. Fiind bazat pe limbajul C/C++, orice program scris folosind SystemC trebuie să aibă şi el o funcţie `main`. Această funcţie nu este vizibilă în implementările de simulari SystemC pentru că librăria deţine propria implementare a funcţiei `main` care la rândul ei apelează funcţia `sc_main` pasându-i parametrii primiţi la apelul programului din linia de comandă. Funcţia `sc_main` va conţine implementarea test bech-ului.

```
int sc_main(int argc, char* argv[]) {
      //ELABORARE
      sc_start(); // <-- Simularea incepe si se termina aici
      //[POST-PROCESARE]
      return COD_EROARE; // Zero indica executie fara erori
}
```

Figura 6. Structura funcției sc_main

Figura 6 ilustrează structura generică a unei implementări de funcție `sc_main` în care se remarcă trei etape principale: elaborare, simulare și post-procesare.

Faza de elaborare este cea în care se pregătește sistemul pentru simulare. În etapa de elaborare se realizează operațiuni precum instanțierea de module sau realizarea de interconexiuni în cadrul sistemului modelat. Pentru instanțierea unui modul ce deține un constructor standard se folosește o sintaxă similară celei de mai jos:

`nume_modul nume_instanta("Instanta1");`.

Se observă aici numele instanței dat ca singur parametru al constructorului. Dacă se dorește obținerea numelui instanței, în interiorul modulului, se poate apela metoda `name()`, moștenită din clasa de bază `sc_module`.

În etapa de simulare se evaluează comportamentul sistemului. În general etapa de simulare presupune doar un apel către funcția `sc_start` care invocă simularea bazată pe comportamentul definit al modulelor. Simularea poate să fie urmată de o etapă de post-procesare în care se evaluează rezultatele simulării sau se generează rapoarte.

Valoarea returnată de funcția `sc_main` este un bun indicator al rezultatului simulării. Din acest motiv se recomandă returnarea unei valori din care să reiasă dacă pe durata simulării modelul a avut comportamentul așteptat.

Exemplul 1. Vom studia în continuare un exemplu în care este implementat un modul simplu cu două threaduri și o metodă. Toate procesele definite în modul afișează un text ce identifică procesul responsabil cu afișarea precum și instanța curentă a modulului. Codul exemplului este redat în Figura 7. Dată fiind simplitatea programului s-a optat pentru realizarea întregii implementări într-un singur fișier sursă. În urma rulării simulării se obsevă ordinea în care procesele sunt apelate de kernelul de simulare. Primele procese apelate sunt metodele din fiecare intanță a modelului. Abia apoi sunt executate threadurile din cele două instanțe.

Exercițiul 1. Decomentați linia comentată din constructorul modulului și apoi rulați din nou simularea după recompilarea codului. Observați și motivați rezultatul obținut.

```
#include "systemc.h"

SC_MODULE(process_example) {
  SC_CTOR(process_example){
    SC_THREAD(Thread1);
    SC_THREAD(Thread2);
    SC_METHOD(Method1);
    //dont_initialize();
  }
  void Thread1(void) {
    cout<<"Thread 1 executat in instanta "<<name()<<endl;
  }

  void Thread2(void) {
    cout<<"Thread 2 executat in instanta "<<name()<<endl;
  }
  void Method1(void) {
    cout<<"Metoda 1 executata in instanta "<<name()<<endl;
  }
};

int sc_main(int argc, char* argv[]) {
  process_example modelInstance1("unu");
  process_example modelInstance2("doi");

  sc_start();

  return 0;
}
```

Figura 7. Exemplul 1 - process_example.cpp

7 Modelarea timpului

SystemC oferă mecanisme pentru modelarea discretă a timpului în simulări. Pentru operațiuni ce necesită manipularea sau evaluarea timpului SystemC pune la dispoziție tipul de date `sc_time`. Timpul este exprimat ca o pereche de valori, prima reprezentând magnitudinea iar a doua unitatea de timp. Unitatea de timp poate să ia una din valorile prezentate în Tabel 1.

Nume macro	**Unitatea de timp**
SC_FS	Femtosecunde
SC_PS	Picosecunde
SC_NS	Nanosecunde
SC_US	Microsecunde
SC_MS	Milisecunde
SC_SEC	Secunde

Tabel 1. Unități de timp disponibile în SystemC

Pentru instanțierea unui obiect de tip sc_time se poate folosi una din următoarele 2 variante de cod:

```
sc_time nume;
sc_time nume(magnitudine, unitate_de_timp);
```

Rezoluţia folosită pentru a măsura trecerea timpului poate să fie precrisă folosind funcţia `sc_set_time_resolution(valoare, unitate_de_timp)`. În lipsa unui apel către funcţia de prescriere a rezoluţiei, sistemul va folosi valoarea implicită a rezoluţiei – 1 picosecundă. Această funcţie trebuie apelată înainte de începerea simulării în secţiunea dedicată elaborării simulării. Prin prescrierea rezoluţiei, orice valoare cu rezoluţie mai bună decât cea specificată va fi rotunjită la cea mai apropiată valoare măsurată în unitatea prescrisă. SystemC defineşte constanta `SC_ZERO_TIME` pentru a desemna valoarea `sc_time(0, SC_SEC)`. După cum vom vedea în secţiunile următoare această constantă are o utilitate specială pentru controlul momentului în timp la care sunt executate procesele. Obiectele de tip `sc_time` pot fi folosite în operaţii aritmetice precum adunarea sau scăderea dar şi în operaţii booleene.

Un aspect important de reţinut este faptul că reprezentarea internă a timpului în SystemC se face folosind intregi pe 64 biţi. Deşi putem să reprezentăm perioade de timp foarte lungi nu trebuie să uităm că există totuşi o limită.

În continuare sunt discutate o serie de funcţii SystemC care folosesc noţiunea de timp.

7.1 sc_start

Funcţia `sc_start` a mai fost menţionată în secţiunea dedicată realizării unui test bench şi a fost utilizată în Exemplul 1 în forma ei fără parametrii. În această formă serveşte la pornirea simulării fără a se specifica o durată de execuţie. O astfel de simulare ar rula la infinit atât timp cât există procese care sunt marcate ca fiind pregătite de execuţie. Dacă se doreşte simularea pentru o durată prestabilită de timp se poate apela funcţia `sc_start` pasând durata de timp ca parametru. Spre exemplu, pentru a simula comportamentul modelului pe o perioadă de o secundă funcţia se poate apela sub forma `sc_time(1, SC_SEC)`. O altă formă de apel este cea în care se oferă ca parametru un obiect de tip sc_time instanţiat şi iniţializat în prealabil.

Timpul simulat stabilit prin parametrii funcţiei `sc_start` nu reflectă durata efectivă a simulării. Aceasta variază în funcţie de complexitatea modelului construit şi de procesele folosite.

7.2 wait

Apelarea funcţiei `wait` este modalitatea prin care threadurile îşi pot suspenda temporar activitatea. Dacă se doreşte suspendarea rulării threadului pentru o durată fixă de timp se apelează funcţia `wait` folosind ca parametru durata suspendării. Funcţia `wait` acceptă ca parametrii fie un obiect preiniţializat de tip `sc_time` fie valoarea directă a duratei de timp. Secvenţa de cod prezentată în Figura 8 ilustrază ambele abordări.

```
void my_model::thread1(void) {
      unsigned int sec = 0;
      for( ; ; ){
            cout<<"Au trecut "<<sec<<" secunde"<<endl;
            sec++;
            wait(1, SC_SEC);
      }
}

void my_model::thread2(void) {
      unsigned int milisec = 0;
      sc_time delay(1, SC_MS);
      for( ; ; ){
            cout<<"Au trecut "<<milisec<<" milisecunde"<<endl;
            sec++;
            wait(delay);
      }
}
```

Figura 8. Utilizarea funcţiei wait pentru suspendarea threadurilor pentru perioade fixe de timp

7.3 sc_time_stamp

Uneori este necesar să monitorizăm trecerea timpului sau să evaluăm timpul de execuţie a anumitor secvenţe din simulare. Pentru astfel de întrebuinţări devine utilă funcţia `sc_time_stamp()` care returnează timpul scurs de la începutul simulării până la apelul ei. Funcţia returnează timpul sub forma unui obiect de tipul `sc_time`. Valoarea returnată va avea cea mai mare unitatea de măsură pentru care se poate obţine o reprezentarea întreagă a valorii timpului. Spre exemplu pentru a reprezenta valoarea de 5000 ns se va returna 5 us sau în cazul în care timpul curent este 0.5 ms va loarea returnată va fi 500 us.

7.4 sc_simulation_time

SystemC pune la dispoziţie timpul curent şi ca o valoare uşor de prelucrat folosind tipuri native C++. Funcţia `sc_simulation_time()` se foloseşte pentru a obţine timpul curent ca o valoare double ce reprezintă un multiplu de unităţi de timp implicite. Stabilirea unităţii de timp implicite la care se referă această funcţie se face apelând funcţia `sc_set_default_time_unit(`*`valoare`*`, `*`unitate de timp`*`)`. Această din urmă poate fi apelată o singură dată, in funcţia `sc_main`, dar numai inainte de pornirea simulării cu `sc_start`.

Exemplul 2. Acest exemplu, al cărui cod sursă este redat în Figura 9. Exemplu 2 process_time_example.cpp Figura 9 urmăreşte execuţia a două threaduri care afişează un text la intervale fixe de timp. Threadurile îşi suspendă execuţia pentru perioade de timp de ordinul milisecundelor, iar timpul total de simulare este limitat la 1 secundă.

Se observă că simularea se termină după o secundă chiar dacă threadul 1 nu şi-a terminat execuţia. De asemenea se remarcă diferenţe în forma de afişare a timpului curent returnat de functia sc_time_stamp. La momentul de timp 500.1 ms

ambele threaduri definite în modul îşi reiau execuţia şi afişează exact acelaşi time stamp. Acest lucru denotă prezenţa paralelismului în execuţia proceselor. În simularea modelului realizat în SystemC concurenţa proceselor este asigurată de kernelul de simulare.

```
#include "systemc.h"

SC_MODULE(process_time_example) {
  SC_CTOR(process_time_example){
    SC_THREAD(Thread1);
    SC_THREAD(Thread2);
  }
  void Thread1(void) {
    wait(500.1, SC_MS);
    cout<<"Thread 1 la 500.1 ms. Timp curent: "<<sc_time_stamp()<<endl;
    wait(500, SC_MS);
    cout<<"Thread 1 la 1000.1 ms. Timp curent: "<<sc_time_stamp()<<endl;
    wait(500, SC_MS);
    cout<<"Thread 1 la 1500.1 ms. Timp curent: "<<sc_time_stamp()<<endl;
  }

  void Thread2(void) {
    wait(250, SC_MS);
    cout<<"Thread 2 la 250 ms. Timp curent: "<<sc_time_stamp()<<endl;
    wait(250.1, SC_MS);
    cout<<"Thread 2 la 500.1 ms. Timp curent: "<<sc_time_stamp()<<endl;
    wait(250, SC_MS);
    cout<<"Thread 2 la 750.1 ms. Timp curent: "<<sc_time_stamp()<<endl;
}
};

int sc_main(int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_US);

  process_time_example modelInstance("time_example");

  sc_start(1, SC_SEC);

  return 0;
}
```

Figura 9. Exemplu 2 process_time_example.cpp

Exerciţiul 2. Modificaţi codul exemplului 2 în aşa fel încât rezoluţia de simulare prescrisă să fie 1 ms. Ce observaţi la rularea simulării? Explicaţi rezultatul obţinut.

Exerciţiul 3. Modificaţi exemplul 2 astfel încât afişarea momentului de timp la care se revine din starea de suspendare să se facă tot timpul în milisecunde. Sugestie: Se va folosi funcţia sc_simulation_time.

8 Concurenţă

Într-un sistem real regăsim o serie de operaţii care sunt executate în paralel. De cele mai multe ori vom întâlni paralelism între operaţiunile efectuate de componente HW diferite, însă, odată cu introducerea microcontrollerelor

multicore în dispozitive embedded, avem de-a face cu paralelism şi la nivelul SW. Concurenţa este deci un aspect important ce trebuie avut în vedere în modelarea sistemelor.

SystemC foloseşte procesele pentru modelarea concurenţei. După cum am arătat în secţiunile anterioare, concurenţa în SystemC nu este reală ci simulată astfel că fiecare proces trebuie să îşi suspende în mod voluntar activitatea pentru a permite execuţia altor procese.

8.1 Evenimente

Funcţionarea kernelului de simulare SystemC este bazată pe evenimente. Un eveniment poate fi definit ca un fenomen instantaneu ce are loc la un moment dat. În SystemC modelarea evenimentelor se face prin utilizarea obiectelor de tip `sc_event`. Declararea unui obiect de tip `sc_event` se face folosind sintaxa: `sc_event` *`nume_eveniment`*;

Apariţia evenimentelor este folosită pentru a declanşa execuţia proceselor. Pentru a face acest lucru posibil, procesul vizat trebuie să aştepte apariţia de evenimente folosind sensitivitate statică sau dinamică. Vom discuta despre aceste două concepte ceva mai târziu în această secţiune.

Generarea unui eveniment se face apelând metoda `notify` a obiectului corespunzător de tip `sc_event`. Notificarea este imediată, atunci când metoda `notify` se apelează fără parametrii, sau temporizată, atunci când primeşte ca parametru durata de timp după care se doreşte să apară notificarea. Spre exemplu, secvenţa `ev.notify(1, SC_US)` face ca evenimentul `ev` să fie declanşat după o microsecundă. Pentru un eveniment poate să existe la un moment dat o singură notificare temporizată. O notificare temporizată existentă poate fi suprascrisă de alta dacă pentru noua notificare se prescrie o durată de timp mai scurtă până la apariţie. Există şi posibilitatea anulării unei notificări temporizate prin apelarea metodei `cancel()`.

8.2 Kernelul de simulare SystemC

Pentru a înţelege mai bine cum să modelăm concurenţa în SystemC trebuie să ştim cum este realizat managementul concurenţei la nivelul kernelului de simulare. Figura 10 ilustrează diagrama de stări a kernelului de simulare SystemC.

Prima etapă a procesului de simulare este iniţializarea. În această fază sunt executate toate procesele definite în faza de elaborare, inainte de apelul funcţiei sc_start() cu excepţia celor pentru care s-a precizat explicit lipsa apelului iniţial implicit prin utilizarea functiei dont initialize(). Ordinea în care are loc execuţia proceselor nu este specificat.

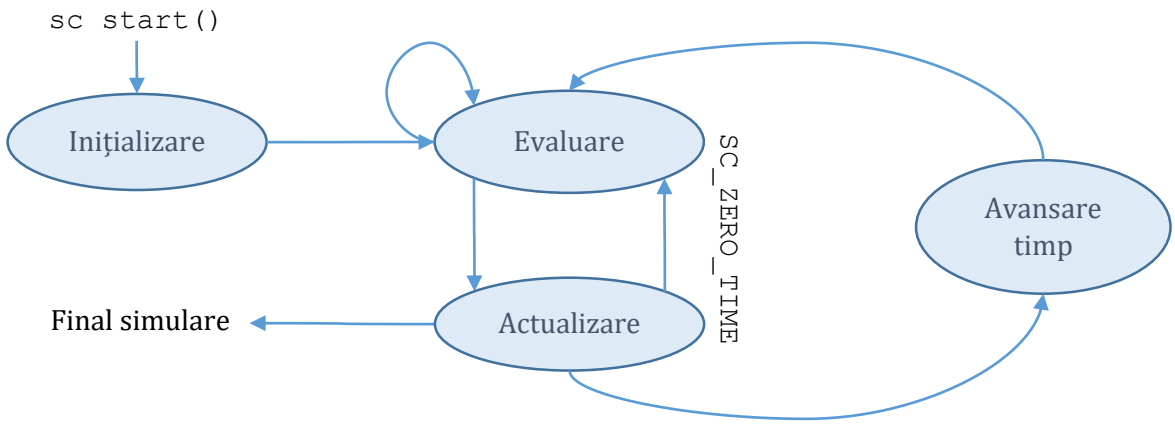

Figura 10. Diagrama de stări a kernelului de simulare SystemC

În etapa de evaluare are loc apelul proceselor marcate ca pregătite pentru execuție în ciclul curent de execuție. Acest proces poate genera marcarea altor funcții ca fiind gata de execuție în același ciclu atunci când au loc notificări imediate. Ciclul de execuție, numit ciclu delta, se termină atunci când toate funcțiile pregătite de execuție au fost executate (inclusiv cele rezultate prin notificări imediate pe durata ciclului curent). Dacă la acest moment nu mai există procese în așteptare de evenimente noi sau a fost atins timpul maxim de execuție setat pentru simulare, simularea se încheie. În caz contrar simularea continuă cu etapa de actualizare în care se actualizează valorile canalelor de comunicare. Urmează execuția proceselor cu temporizare delta, dacă acestea există, sau avansarea la următorul moment al simulării.

Pentru a descrie procesele temporizate delta trebuie să introducem un caz particular de notificări. Acesta se referă la notificările care folosesc apelul cu parametru pentru notificări temporizate setând pe 0 durata de temporizare, de exemplu: `e.notify(SC_ZERO_TIME)` sau `e.notify(0)`. Aceste notificări nu sunt considerate notificări imediate, prin urmare nu sunt procesate în ciclul de evaluare în care apar. Pentru că nu este nevoie de trecerea la un alt interval de timp aceste procese se vor procesa într-un ciclu delta suplimentar fără a se trece prin etapa de avans temporal. În aceeași categorie intră și procesele de tip thread care apelează funcția `wait` cu temporizare `SC_ZERO_TIME`.

Avansarea timpului de simulare constă în trecerea la următorul moment în timp pentru care au fost înregistrate notificări sau care este așteptat (apel al funcției `wait`). Procesele aflate în așteptarea acestui moment de timp sunt marcate pentru execuție după care se începe un nou ciclu de evaluare.

8.3 Sensitivitate statică

Sensitivitatea statică a proceselor se referă la stabilirea înaintea începerii simulării a unor parametrii care sunt folosiți pentru a stabili momentul în care are loc

execuţia unui proces. Parametrii de sensitivitatea statică prescrişi nu pot să fie schimbaţi în timpul rulării. Există însă posibilitatea de a îi suprascrie.

Stabilirea sensitivităţii statice se realizează folosind metoda `sensitive` printr-un apel clasic în care se pasează ca parametru evenimentul (sau lista de evenimente) pentru care se aplică sau, ca alternativă, folosind operatorul stream: `sensitive << eveniment`. Sintaxa întâlnită cel mai des este aceasta din urmă.

Pentru fiecare proces stabilirea sensitivităţii statice se va face în constructorul modulului imediat după înregistrarea procesului corespunzător şi se aplică oricărui tip de proces.

Exemplul 3. Urmăriţi exemplul următor în care se simulează comportamentul unui bistabil de tip D. Implementarea simulării prezentată în figurile 11-13 este de data aceasta realizată în fişiere diferite pentru definiţia modulului, implementarea acestuia şi test bench. Pentru compilare se vor menţiona în lista de fişiere sursă ambele fişiere .cpp ca în exemplul următor:

```
g++ -I. -I$SYSTEMC_HOME/include -L. -L$SYSTEMC_HOME/lib-linux -Wl,-rpath=$SYSTEMC_HOME/lib-linux -o bistabil_d testbench_bistabil_d.cpp bistabil_d.cpp -lsystemc -lm
```

```
#include <systemc.h>

SC_MODULE(BistabilD) {
  bool d;
  bool q;
  bool clk;
  sc_event update_data_event;

  SC_HAS_PROCESS(BistabilD);

  BistabilD (sc_module_name name);
  void update_data_method();
  void clk_thread();
  void toggle_d_thread();
};
```

Figura 11. Exemplul 3 - bistabil_d.h

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

int sc_main(int argc, char **argv)
{
  BistabilD b("Bistabil_1");

  sc_start(5, SC_MS);

  return 0;
}
```

Figura 12. Exemplul 3 - testbench_bistabil_d.cpp

```
#include "bistabil_d.h"

BistabilD:: BistabilD (sc_module_name name): sc_module(name)
{
  d = 0;
  q = 0;
  clk = 0;
  SC_THREAD(clk_thread);
  SC_THREAD(toggle_d_thread);

  SC_METHOD(update_data_method);
  dont_initialize();
  sensitive << update_data_event;
}

void BistabilD::update_data_method()
{
  d=!d;
  cout << sc_time_stamp() << ": D = " << d << endl;
}

void BistabilD::clk_thread()
{
  for (;;) {
    clk=!clk;
    if(clk) {
      //wait(SC_ZERO_TIME);
      q = d;
    }
    cout << sc_time_stamp() << ": Q = " << q << endl;
    wait(0.5, SC_MS);
  }
}

void BistabilD::toggle_d_thread()
{
  for (;;) {
    update_data_event.notify();
    wait(1, SC_MS);
  }
}
```

Figura 13. Exemplul 3 - bistabil_d.cpp

Bistabilul de tip D are ca intrări linia de date D şi semnalul de tact, iar ca ieşiri liniile Q şi $\bar{Q}$. Modelul realizat reprezintă semnalele de intrare şi ieşire ca variabile boolene. Pentru simplitate s-a renunţat la reprezentarea ieşirii Q negate. Semnalul de tact cu perioadă de 1 ms este generat cu ajutorul unui thread. Schimbarea valorii D are loc într-o metodă ce are înregistrat în lista de sensitivitate un eveniment declanşat periodic de un al doilea thread .

Rulând simularea puteţi observa schimbările efectuate asupra semnalului de intrare şi efectul pe care aceste modificări îl au asupra ieşirii bistabilului. Se remarcă faptul ca la momente de timp multipli de 1 ms are loc întâi afişarea ieşirii Q urmată apoi de actualizarea valorii D. Acest comportament este cauzat de modelul de management al kernelului de simulare. În aceste puncte cele două

threaduri sunt marcate ca fiind pregătite de execuţie, dar metoda care actualizează intrarea D nu este încă gata de execuţie pentru că nu a fost lansată încă notificarea pentru evenimentul update_data_event. Abia după execuţia threadului toggle_d_thread are loc notificarea şi adăugarea metodei în lista de proceduri pregătite pentru execuţie, motiv pentru care actualizarea intrării D nu afectează ieşirea Q.

Exerciţiul 4. Modificaţi codul exemplului anterior în aşa fel încât actualizarea intrării D a bistabilului să aibă loc înainte de actualizarea ieşirii Q.

8.4 Sensitivitate dinamică

Procesele îţi pot specifica în mod dinamic sensitivitatea. Modul în care se realizează specificarea dinamică a sensitivităţii diferă în funcţie de tipul procesului. Astfel, procesele de tip `SC_THREAD` vor folosi metoda `wait`, iar cele de tip `SC_METHOD` vor folosi metoda `next_trigger`. În cele ce urmează vom detalia pe rând fiecare din cele două cazuri.

Despre metoda `wait` am mai discutat într-una din secţiunile anterioare amintind de utilizarea ei pentru a obţine suspendarea funcţionării threadului pe o perioadă de timp specificată. Această metodă poate fi folosită şi sub alte forme în funcţie de tipul parametrilor pe care îi primeşte. Variantele de sintaţă pentru apelul metodei wait sunt redate în Figura 14.

```
wait(timp);
wait(eveniment);
wait(eveniment1 | eveniment2 …); //oricare eveniment
wait(eveniment1 & eveniment2 …); //toate evenimentele
wait(timeout, eveniment); // eveniment cu timeout
wait(timeout, eveniment1 | eveniment2…);//oricare eveniment cu timeout
wait(timeout, eveniment1 & eveniment2 …);//toate evenimentele cu timeout
wait(); // sensitivitate statica
```

Figura 14. Variante de sintaxă pentru utilizarea metodei wait

Pasarea de argumente de tip `sc_event` la apelul funcţiei `wait` face ca threadul din care s-a făcut apelul să declare sensitivitatea pentru evenimentul primit ca parametru. Operatorul `|` aplicat evenimentelor face ca threadul execuţia threadului să fie reluată la apariţia oricăruia din evenimentele cărora li se aplică. Este important de reţinut faptul că nu există posibilitatea de a afla care din evenimente a fost generat. Dacă se doreşte acest lucru trebuie implementat un mecanism dedicat. Similar, aplicarea operatorului `&` face ca reluarea execuţiei să se facă atunci când toate evenimentele menţionate au loc. În această situaţie nu se poate şti ordinea în care au apărut evenimentele.

Variantele de apel cu timeout sunt utile atunci când există posibilitatea ca evenimentele aşteptate de thread să nu apară, evitându-se astfel situaţia nedorită în care threadul rămâne blocat. Pentru a afla dacă reluarea execuţiei s-a făcut ca

urmarea a unui timeout se apelează metoda `timed_out()` imediat după revenirea din apelul metodei `wait`. De obicei astfel de apeluri ale funcţiei wait se folosesc în modelarea şi simularea protocoalelor sau a unor potenţiale condiţii de eroare. Dacă threadul foloseşte surse de sensitivitate statică atunci apelând metoda wait fără parametrii se va obţine suspendarea execuţiei până la apariţia unuia din evenimentele declarate ca sensitivitate statică.

Similar proceselor thread, procesele de tip `SC_METHOD` folosesc pentru a specifica sensitivitatea dinamică metoda `next_trigger` cu aceleaşi variante de parametri ilustrate în Figura 15.

Spre deosebire de metoda wait care suspenta threadul, metoda `next_trigger` doar acţionează asupra listei de sensitivitate corespunzătoare metodei. Pe parcurul execuţiei unei metode pot să apară mai multe apeluri către next _trigger care pot suprascrie efecte ale apelurilor anterioare.

```
next_trigger(timp);
next_trigger(eveniment);
next_trigger(eveniment1 | eveniment2 …); //oricare eveniment
next_trigger(eveniment1 & eveniment2 …); //toate evenimentele
next_trigger(timeout, eveniment); // eveniment cu timeout
next_trigger(timeout, eveniment1 | eveniment2…);//oricare eveniment cu
//timeout
next_trigger (timeout, eveniment1 & eveniment2 …);//toate evenimentele cu
//timeout
next_trigger (); // sensitivitate statica
```

Figura 15. Variante de sintaxă pentru utilizarea metodei next_trigger

Exerciţiul 5. Modificaţi codul exemplului 3 în aşa fel încât să folosească doar sensitivitate dinamică pentru declanşarea execuţiei tuturor proceselor

9 Comunicare între module

Rezultatul modelării unui sistem in SystemC constă într-o structură ierarhică de module în care apare necesitatea transmiterii de informaţii. Până în acest punct singura formă de comunicare prezentată a fost notificarea apariţiei evenimentelor. Există totuşi situaţii în care sunt necesare alte mecanisme pentru transmiterea de informaţii. Un bun exemplu este cazul în care un eveniment poate să fie declanşat de mai multe module. Folosind doar evenimente, modulul sensibil la declanşarea lor nu va putea afla care componentă a modelului este responsabilă de această declanşare. O soluţie, în aceste cazuri, ar fi utilizarea unor mecanisme native C++ însă această abordare nu va putea beneficia de funcţionalităţi specifice întroduse de SystemC pentru modelarea şi simularea comunicării.

SystemC introduce două mecanisme pentru a facilita comunicarea între module: *canale* şi *porturi*. Aceste mecanisme sunt tratate pe rând în cele ce urmează.

9.1 Canale

În SystemC canalele oferă suportul pentru comunicarea între module. Librăria pune la dispoziţie mai multe tipuri de canale predefinite dar şi posibilitatea de a implementa canale cu comportament specific. În această secţiune ne vom referi la canale ierarhice care folosesc principiul *evaluare-actualizare*. Denumirea acestui principiu face referire directă la două etape din bucla principală de procesare a kernelului SystemC. Utilizarea acestui tip de canal oferă fiabilitate atunci când canalul este folosit în acelaşi ciclu delta atât pentru operaţii de citire cât şi de scriere. Pentru a evita apariţia unor conflicte, în cadrul fazei de evaluare dintr-un ciclu delta toate operaţiile de citire a canalului vor avea ca rezultat valoarea canalului de la începutul ciclului. Operaţiile de scriere într-un canal vor afecta valoarea canalului doar în etapa de actualizare ce urmează celei de evaluare din ciclul delta.

Unul din canalele de bază în SystemC care folosesc paradigma evaluare-actualizare este `sc_signal`. Pentru instanţierea unui semnal folosim sintaxa următoare: `sc_signal<`*`tip_data`*`> nume_semnal;`. Putem deci crea semnale care să reprezinte diverse tipuri de date. Scrierea şi citirea valorii semnalului se face apelând metoda `write(`*`valoare`*`)`, respectiv `read()`. Ca urmare a apelării metodei `write` are loc salvarea viitoarei valori şi înregistrarea unei cereri de actualizare a valorii semnalului. Actualizarea va avea loc în următorul ciclu de actualizare.

Obiectele de tip `sc_signal` deţin un eveniment a cărui declanşare are loc atunci când se schimbă valoarea semnalului. Pentru accesarea evenimentului se apelează metoda `default_event()` a semnalului. Ca orice evenimente, cele asociate semnalelor se pot utiliza pentru a stabili sensitivitatea statică sau dinamică a unui proces ca în exemplele din Figura 16.

```
sc_signal<bool> semnal;
sensitive << semnal.default_event();
wait(semnal.default_event());
next_trigger(semnal.default_event());
```

Figura 16. Exemple de utilizare a evenimentului asociat unui semnal

9.2 Porturi

Pentru a realiza comunicarea între module prin intermediul unui canal avem nevoie de porturi. Porturile nu sunt altceva decât intrările şi ieşirile unui modul. Două din tipurile de bază de porturi utilizate în SystemC sunt `sc_in<`*`tip_data`*`>` `sc_out<`*`tip_data`*`>`. Porturile sunt tot timpul definite în cadrul definiţiei unui modul şi pot fi interpretate ca fiind pointeri către canale.

Pentru a putea asigura comunicarea între module porturile trebuie să fie conectate la canale în faza de instanţiere a modulelor. Această operaţiune poate să fie

realizată folosind principiul de *conectare după nume* sau cel de *conectare după poziţie*. Pentru a ilustra cele două abordări folosim exemplul unei simulări în care un modul cu o intrare şi o ieşire ce implementează un proces oarecare foloseşte un canal extern setat ca buclă de feedback. Definiţia modulului este schiţată în Figura 17. În faza de elaborare a test bench-ului este creat canalul folosit pentru feedback şi se instanţiază modulul. Pentru conectarea după nume a porturilor prin canal se foloseşte sintaxa sugerată în Figura 18 în care pentru fiecare port se specifică separat canalul la care este conectat. Când folosim această abordare nu este importantă ordinea în care se realizează conectarea porturilor. La conectarea după poziţie, folosind sintaxa prezentată în Figura 19, se conectează într-un singur pas toate porturile unui modul. În acest caz conectarea porturilor la canalele menţionate ca parametrii are loc în ordinea în care acestea au fost declarate în definiţia modulului, deci trebuie asigurată corespondenţa corectă între porturi şi lista de canale.

```
SC_MODULE(proces){
  sc_in<bool> intrare;
  sc_out<bool> iesire;

  SC_CTOR(proces){
    ...
  }
  ...
};
```

Figura 17. Definiţia unui modul ce implementează un proces

```
int sc_main(int argc, char **argv)
{
  sc_signal<bool> feedback;

  proces proces1("Proces");

  proces1.intrare(feedback);
  proces1.iesire(feedback);
  ...
}
```

Figura 18. Conectarea după nume a porturilor printr-un canal

```
int sc_main(int argc, char **argv)
{
  sc_signal<bool> feedback;

  proces proces1("Proces");

  proces1(feedback, feedback);
  ...
}
```

Figura 19. Conectarea după poziţie a porturilor printr-un canal

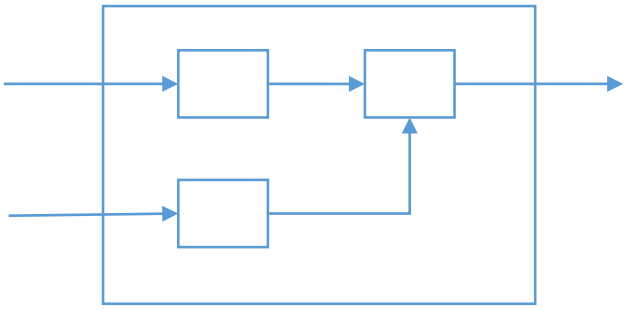

Figura 20. Submodule conectate direct la porturile modulului părinte

Un caz special este cel în care semnalul corespunzător unui port al modulului părinte este oferit direct ca intrare sau obţinut ca ieşire a unui modul din structura internă a modulului părinte. O astfel de situaţie este ilustrată în Figura 20. În acest caz, pentru realizarea conexiunilor între porturile submodulelor şi cele ale modulului părinte nu se foloseşte un canal. Se vor folosi numele porturilor modulului părinte în faza de conectare a porturilor aferente submodulelor ca în exemplul din Figura 21.

```
SC_MODULE(submodul){
  sc_in<bool> intrare_submodul;
  sc_out<bool> iesire_submodul;

  SC_CTOR(proces){
    ...
  }
  ...
};

SC_MODULE(proces){
  sc_in<bool> intrare;
  sc_out<bool> iesire;

  submodul submodul1;

  SC_CTOR(proces){
    submodul1.intrare_submodul(intrare);
    submodul1.iesire_submodul(iesire);
    ...
  }
  ...
};
```

Figura 21. Conectarea porturilor unui modul părinte la porturile submodulelor

În faza de elaborare se pot seta valori de iniţializare pentru porturile de ieşire apelând metoda `initialize()` în cadrul constructorului. Astfel pentru a iniţializa canalul conectat la un port de ieşire vom folosi sintaxa *`port_iesire`*`.initialize(`*`valoare`*`)`.

Prin intermediul porturilor se pot efectua operaţii de citire şi scriere asupra canalelor la care acestea sunt conectate. Acest lucru este posibil prin apelarea metodelor `read()` şi `write(`*`valoare`*`)`, corespunzătoare unui canal, prin

intermediul portului folosind operatorul –>. Astfel pentru citire vom folosi sintaxa `nume_port->read()`, iar pentru scriere `nume_port->write(valoare)`. Pentru aceleaşi operaţii de scriere şi citire instanţa unui port se poate folosi ca o variabilă normală. Totuşi se recomandă utilizarea celor două metode dedicate pentru a uşura înţelegerea codului şi a evita eventuale confuzii. Indiferent de abordare trebuie reţinut faptul că în urma unei operaţii de scriere/atribuire valoarea canalului nu se va schimba imediat ci în urma următorului ciclu de actualizare.

În lista de sensitivitate declarată în constructorul unui modul se poate specifica şi sensitivitatea la schimbarea valorii unui port. SystemC permite declararea unei astfel de sensitivităţi prin specificarea numelui portului doar în cazul în care canalul aferent oferă o implementare a metodei `default_event()`.

Un alt tip de port similar cu cele două deja prezentate este `sc_inout<tip_data>`. Spre deosebire de porturile dedicate de intrare sau ieşire un port de tip `sc_inout` poate să fie folosit atât pentru citirea din cât şi pentru scrierea în canalul la care este conectat.

Exemplul 4. Exemplul următor ilustrează utilizarea porturilor şi a canalelor prin implementarea unui divizor de frecvenţă (f/2) bazat pe bistabie de tip D conform schemei din Figura 22. Generarea semnalului de clock precum şi monitorizarea ieşirii sunt realizate în cadrul modulului divizor de frecvenţă.

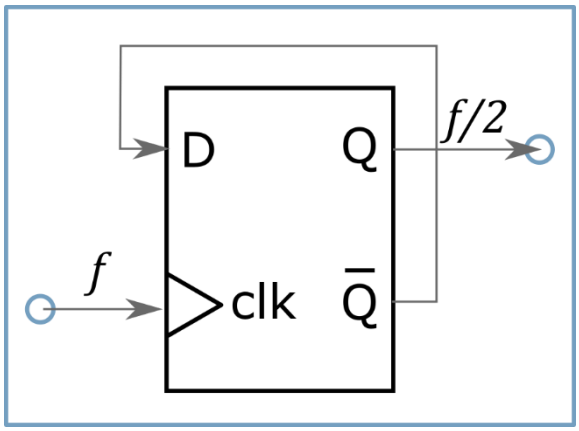

Figura 22. Schema unui divizor de frecvenţă

```
#include <systemc.h>

SC_MODULE(BistabilD) {
  sc_in<bool> d;
  sc_in<bool> clk;
  sc_out<bool> q;
  sc_out<bool> not_q;

  SC_HAS_PROCESS(BistabilD);

  BistabilD (sc_module_name name);
  void update_output_method();
};
```

Figura 23. Exemplul 4 –bistabil_d.h

Funcţionalitatea bistabilului este implementată ca modul separat având porturi declarate pentru fiecare din intrările şi ieşirile aferente. Modulul divizor de tensiune instanţiază un modul de tip bistabil D şi generează semnalul de clock, cu o perioadă de 1ms (frecvenţă de 1kHz), de la intrarea acestuia. Divizorul realizează de asemenea conexiunea de loopback între ieşirea $\bar{Q}$ şi intrarea D, prin intermediul unui canal, şi monitorizează semnalul de la ieşirea Q. Implementările modulelor şi a testbench-ului sunt redate în figurile 23-27. La rularea simulării se poate observa că semnalul de clock rezultat are perioada de 2ms echivalent unei frecvenţe de 0,5kHz.

```
#include "bistabil_d.h"

BistabilD:: BistabilD (sc_module_name name): sc_module(name)
{
  q.initialize(0);
  not_q.initialize(0);

  SC_METHOD(update_output_method);
  sensitive << clk;
}

void BistabilD::update_output_method()
{
  if(clk.read() == 1){
    q.write(d.read());
    not_q.write(!d.read());
    cout << sc_time_stamp() << ": D = " << d << ", Q = " << q << endl;
  }
}
```

Figura 24. Exemplul 4 –bistabil_d.cpp

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

SC_MODULE(DivizorFrecventa) {
  sc_signal<bool> loopback_sig;
  sc_signal<bool> in_clk_sig;
  sc_signal<bool> out_clk_sig;

  BistabilD bistabil;

  SC_HAS_PROCESS(DivizorFrecventa);

  DivizorFrecventa(sc_module_name name);
  void print_output_method();
  void clk_thread();
};
```

Figura 25. Exemplul 4 –divizor_frecventa.h

```
#include "divizor_frecventa.h"

DivizorFrecventa::      DivizorFrecventa      (sc_module_name      name):
sc_module(name), bistabil("BistabilD")
{
  bistabil.d(loopback_sig);
  bistabil.q(out_clk_sig);
  bistabil.clk(in_clk_sig);
  bistabil.not_q(loopback_sig);

  SC_THREAD(clk_thread);

  SC_METHOD(print_output_method);
  sensitive << out_clk_sig.default_event();
}

void DivizorFrecventa::print_output_method()
{
  cout << sc_time_stamp() << ": Out = " << out_clk_sig.read() << endl;
}

void DivizorFrecventa::clk_thread()
{
  for (;;) {
    wait(0.5, SC_MS);
    in_clk_sig.write(!in_clk_sig.read());
  }
}
```

Figura 26. Exemplul 4 –divizor_frecventa.cpp

Exerciţiul 6. Modificaţi codul exemplului 4 în aşa fel încât semnalul de clock de intrare să fie generat de un modul separat numit generator, iar afişarea informaţiilor despre semnalul clock de ieşire să fie realizate de un modul supervizor. Ambele module nou adăugate se vor instanţia şi interconecta cu

bistabilul prin intermediul modulului divizor de frecvenţă conform exemplului din Figura 28.

Exerciţiul 7. Efectuaţi modificări adiţionale modulului divizor de frecvenţă pentru a-l face să genereze la ieşire frecvenţa de intrare divizată cu 4.

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

int sc_main(int argc, char **argv)
{
  DivizorFrecventa divizor("Divizor_f/2");

  sc_start(5, SC_MS);

  return 0;
}
```

Figura 27. Exemplul 4 –testbench_divizor_frecventa.cpp

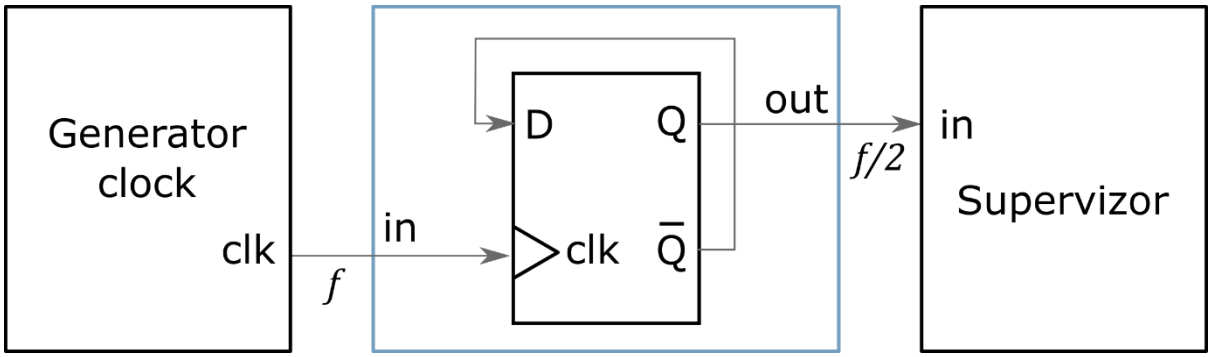

Figura 28. Divizor de frecvenţă cu generator extern de clock

10 Debug prin vizualizarea semnalelor

În multe cazuri când analizăm comportamentul sistemului modelat este dificil să urmărim evoluţia semnalelor folosind doar tipărirea unor informaţii în consolă. Pentru urmărirea formelor de undă este mai potrivită utilizarea unor aplicaţii dedicate.

SystemC nu oferă funcţionalităţi de vizualizare a semnalelor, dar poate să exporte date într-un format compatibil cu aplicaţii specializate în vizualizarea formelor de undă cum ar fi GTKWave. Formatul folosit pentru exportarea datelor din SystemC este VCD (Value Change Dump)

10.1 Exportarea datelor în format VCD

Pentru exportarea datelor trebuie să folosim un obiect de tip `sc_trace_file`. Pentru a crea un obiect de acest tip SystemC pune la dispoziţie funcţia `sc_create_vcd_trace_file(`*`nume_fisier_trace`*`)` care primeşte ca parametru un şir de caractere conţinând numele fişierului de ieşire (fără extensia

vcd) şi va returna un pointer la obiectul creat. Înainte de ieşirea din program trebuie să ne asigurăm că am închis fişierul de trace apelând funcţia `sc_close_vcd_trace_file(fisier_trace)`.

```
SC_MODULE(DivizorFrecventa) {
  sc_trace_file* tracefile;

  sc_signal<bool> loopback_sig;
  sc_signal<bool> in_clk_sig;
  sc_signal<bool> out_clk_sig;

  BistabilD bistabil;

  SC_HAS_PROCESS(DivizorFrecventa);

  ~DivizorFrecventa();
  ...
};
```

Figura 29. Modificare header divizor de frecvenţă pentru salvarea semnalelor într-un fişier de trace

Pentru a adăuga semnale în lista de trace a fişierului folosim funcţia `sc_trace` cu sintaxa `sc_trace(fisier_trace, nume_semnal, "nume_semnal")`. Funcţia primeşte ca parametrii obiectul corespunzător fişierului de trace, numele obiectului semnal ce trebuie inclus în trace şi un şir de caractere reprezentând textul corespunzător semnalului ce va fi afişat în aplicaţia de vizualizare a semnalelor. Apelul funcţiei `sc_trace` trebuie făcut după ce au fost definite semnalele adăugate. Apelarea acestei funcţii face ca semnalul menţionat să fie "urmărit" pe perioada simulării, în fişierul de trace fiind salvate valorile semnalului la intervale de timp date de rezoluţia timpului folosită în simulare.

```
DivizorFrecventa::      DivizorFrecventa      (sc_module_name      name):
sc_module(name), bistabil("BistabilD")
{
  tracefile = sc_create_vcd_trace_file("wave");

  if (!tracefile){
    cout <<"There was an error."<<endl;
  }
  else {
    sc_trace(tracefile, in_clk_sig, "in_clk");
    sc_trace(tracefile, loopback_sig, "loopback");
    sc_trace(tracefile, out_clk_sig, "out_clk");
  }
...
}
...
DivizorFrecventa::~DivizorFrecventa ()
{
    sc_close_vcd_trace_file(tracefile);
}
```

Figura 30. Modificare fişier sursă divizor de frecvenţă pentru salvarea semnalelor într-un fişier de trace

Pentru a ilustra modul în care se realizează exportarea semnalelor ne folosim de exemplul anterior al divizorului de frecvenţă. Figurile Figura 29 şi Figura 30 prezintă modificările aduse modulului divizor de frecvenţă pentru a permite salvarea variaţiei fiecărui semnal folosit în cadrul modulului. Se observă utilizarea constructorului pentru crearea obiectului corespunzător fişierului de trace şi adăugarea semnalelor în lista de semnale monitorizate. De asemenea, se remarcă închiderea fişierului de trace în destructorul modulului.

10.2 GTKWave

Fişiere şi informaţii specifice necesare pentru instalarea pe sisteme de operare Windows, Linux şi MacOS sunt disponibile pe site-ul GTKWave: http://gtkwave.sourceforge.net/. În continuare vom ilustra paşii necesari instalării pe un sistem Linux Ubuntu.

1. Descărcaţi şi dezarhivaţi sursele, după care deschideţi un terminal şi navigaţi în folderul ce conţine fişierele sursă.
2. Asiguraţi-vă că sunt instalate dependinţele necesare pentru compilarea aplicaţiei GTKWave. Dacă acestea lipsesc, instalaţi-le folosind comanda:

```
$ sudo apt-get install tcl-dev tk-dev gperf libgtk2.0-dev
```

3. Porniţi configurarea, compilarea şi instalarea aplicaţiei

```
$ sudo ./configure --disable-xz
$ sudo make
$ sudo make install
```

Pentru a deschide un fişier de tip VCD folosim următoarea sintaxă în terminal:

```
$ gtkwave nume_fisier.vcd
```

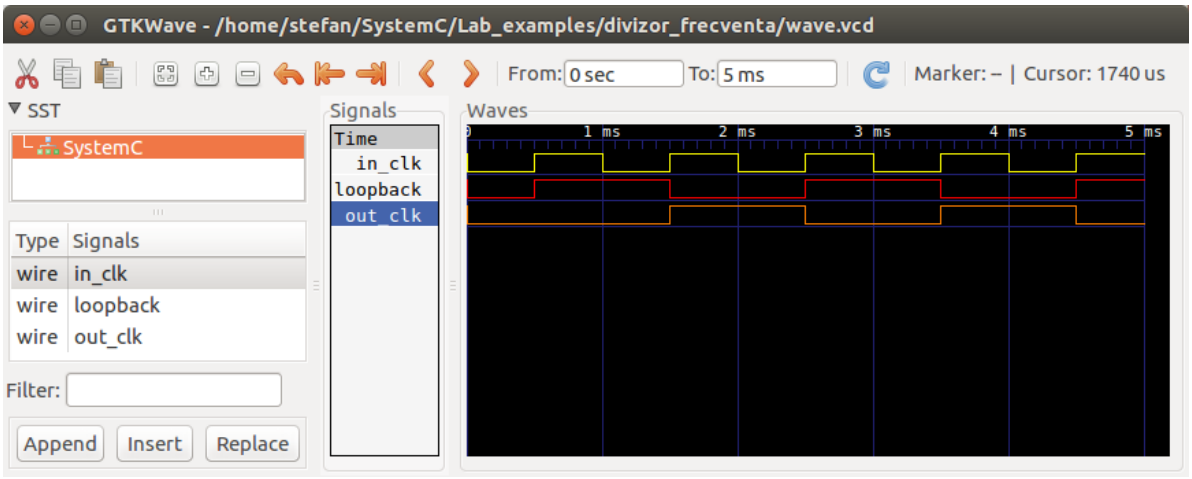

Figura 31. Formele de undă ale semnalelor folosite în exemplul divizorului de frecvenţă

La pornirea programului semnalele nu sunt încărcate implicit în lista de semnale vizualizate. Acestea trebuie selectate manual și adăugate în câmpul `Signals` adiacent elementului de vizualizare `Waves`. În Figura 31 este prezentată variația semnalelor din implementarea divizorului de frecvență vizualizată folosind GTKWave. Mai multe informații despre această aplicație și funcționalitățile pe care le pune la dispoziție găsiți în ghidul de utilizare GTKWave [5]

Exercițiul 8. Adăugați funcționalitatea de salvare a semnalelor în exemplul 4 și vizualizați semnalele folosind GTKWave.

Referințe

[1] Open SystemC Initiative, „Functional Specification for SystemC 2.0 - Update for SystemC 2.0.1," 2002.

[2] Open SystemC Initiative, „Draft Standard SystemC Language Reference Manual," 2005.

[3] Open SystemC Initiative, „SystemC Version 2.0 User's Guide - Update for SystemC 2.0.1," 2002.

[4] J. Donovan și D. C. Black, SystemC: From The Ground Up, Eclectic Ally, 2004.

[5] „GTKWave 3.3 Wave Analyzer User's Guide," 13 January 2018. [Interactiv]. Available: http://gtkwave.sourceforge.net/gtkwave.pdf. [Accesat 25 February 2018].