1.7 SystemC примитивни канали

SystemC има уграђене механизме који олакшавају моделовање комуникације. Ови механизми називају се каналима. Постоје две врсте канала: примитивни и хијерархијски. У овом поглављу биће описани примитивни канали.

SystemC примитивни канали не садрже хијерархију, немају процесе и дизајнирани су да буду брзи и једноставни. Сви примитивни канали наслеђују базну класу `sc_prim_channel`. У овом поглављу биће описани: `sc_mutex`, `sc_semaphore`, `sc_fifo` и `sc_signal`.

1.7.1 sc_mutex

Мутекс је термин који долази из енглеског језика (Mutual Exclusion) и значи узајамно искључивање. Сваки процес коме је потребан дељени ресурс, мора закључати мутекс пре приступања ресурсу. Када ресурс више није потребан процесу, мутекс је потребно откључати. Уколико процес покуша да приступи ресурсу кроз закључан мутекс, процес ће бити спречен у тој акцији.

SystemC садржи класу која имплементира мутекс, `sc_mutex` примитивни канал. Класа садржи блокирајуће и неблокирајуће методе за манипулисање мутексом. Блокирајуће методе не смеју се користити у `SC_METHOD` процесима, пошто се ти процеси морају извршити од почетка до краја, без чекања.

Методе за рад са мутексом су:

```
int lock();
int trylock();
int unlock();
```

Метода `lock` закључава мутекс. Она је блокирајућа. То значи да када се ова метода позове, уколико је мутекс закључан, процес ће чекати док се мутекс не откључа и потом ће пробати да закључа мутекс и добије приступ дељеном ресурсу.

Метода `trylock` исто закључава мутекс али је неблокирајућа. То значи да ће процес пробати да закључа мутекс. Уколико је он већ закључан, ова метода не блокира процес већ враћа повратну вредност -1. Уколико је мутекс откључан ова метода га закључава и враћа вредност различиту од -1.

Метода `unlock` откључава мутекс. Уколико мутекс није закључан од процеса који позива ову методу, повратна вредност је -1, у супротном је различита од -1.

Једноставан пример који користи мутекс, може се релизовати једним модулом који користит мутекс за синхронизацију (листинг 1.10). Тај модул могао би се инстацирати два пута унутар главног програма.

Листинг 1.10: Илустрација мутекса

```
#include <systemc>
#include <iostream>

using namespace sc_core;

SC_MODULE(BusGen)
{
public:
        SC_HAS_PROCESS(BusGen);

        BusGen(sc_module_name name, sc_mutex* );

protected:
        sc_mutex* m;

        void proc();
};

BusGen::BusGen(sc_module_name name, sc_mutex* m) : sc_module(name), m(m)
{
        SC_THREAD(proc);
}

void BusGen::proc()
{
        while(1)
        {
                wait(5, SC_NS);
                m->lock();
                std::cout << "@ " << sc_time_stamp() <<
                        " bus locked by " << name() << std::endl;
                wait(3, SC_NS);
                m->unlock();
        }
}

int sc_main(int argc, char* argv[])
{
        sc_mutex mtx;
        BusGen g1("g1", &mtx);
        BusGen g2("g2", &mtx);

        sc_start(100, SC_NS);

    return 0;
}
```

Модул који приступа дељеном ресурсу зове се `BusGen`. Садржи један процес `proc`. Конструктор овог модула прима показивач на класу `sc_mutex`. У главном програму инстанцирана су два модула `BusGen` и повезана су истом `sc_mutex` променљивом.

У свом процесу, модули `BusGen` чекају 5 ns и затим покушавају да приступе дељеном ресурсу методом `lock` мутекса. Само један од њих добиће приступ у једном тренутку. Дељени ресурс у овом примеру је код између позива метода `lock` и `unlock`. Оваква заштита на делу кода назива се критична секција.

1.7.2 sc_semaphore

Неки ресурси допуштају да им већи број објеката приступи. За моделовање таквих случајева потребан нам је мутекс, који броји колико објеката је приступило дељеном ресурсу. Семафор је такво поопштење мутекса. Може се рећи и да је мутекс семафор који броји до 1. Да би се моделовали такви случајеви SystemC обезбеђује класу `sc_semaphore`.

Методе класе `sc_semaphore` су:

```
int wait();
int trywait();
int post();
int get_value();
```

Метода `wait` је блокирајућа. Уколико је вредност семафора 0, метода ће обуставити извршавање процеса, све док вредност семафора не повећа неки други процес. У том тренутку, процес ће наставити своје извршавање, пробаће да добије приступ дељеном ресурсу и пробаће да смањи вредност семафора.

Метода `trywait` је неблокирајућа. Уколико је вредност семафора 0, метода `trywait` без блокирања вратиће вредност -1 и неће модификовати вредност семафора. У суптроном, процес ће добити приступ дељеном ресурсу и вредност семафора смањиће се за 1.

Метода `post` повећаће вредност семафора. Уколико постоје процеси који чекају да се вредност семафора повећа, тачно један процес смањиће вредност семафора и добиће приступ док ће остали процеси остати блокирани. Одабир процеса који ће добити приступ је недерминистички.

Метода `get_value` враћа вредност семафора.

Леп пример за употребу семафора био би ситуација у којој велики број аутомобила чека на мањи број паркинг места (листинг 1.11).

Листинг 1.11: Илустрација семафора

```
#include <systemc>
#include <iostream>
#include <list>
#include <string>

using namespace sc_core;

SC_MODULE(Car)
{
public:
        SC_HAS_PROCESS(Car);

        Car(sc_module_name name, sc_semaphore* );

protected:
        sc_semaphore* s;

        void proc();
};
```

```
Car::Car(sc_module_name name, sc_semaphore* s) : sc_module(name), s(s)
{
        SC_THREAD(proc);
}

void Car::proc()
{
        wait(rand()%17 + 3, SC_NS);
        while(1)
        {
                s->wait();
                std::cout
                        << "@ " << sc_time_stamp() << " "
                        << name() << " entered. Places left "
                        << s->get_value() << std::endl;
                wait(rand()%37 + 3, SC_NS);
                s->post();
        }
}

int sc_main(int argc, char* argv[])
{
        const int NUMBER_OF_PLACES = 50;

        sc_semaphore park(NUMBER_OF_PLACES);

        std::list<Car*> cars;
        for (int i = 0; i < 100; ++i)
        {
                std::string instName = "Car" + std::to_string(i);
                Car* c = new Car(instName.c_str(), &park);
                cars.push_back(c);
        }

        sc_start(200, SC_NS);

        std::list<Car*>::iterator it;
        for (it = cars.begin(); it != cars.end(); ++it)
                delete *it;

    return 0;
}
```

Модул `Car` моделује понашање кола. Семафор моделује понашање паркинга. Конструктор модула `Car` прима показивач на семафор као параметар. Модул има само један процес. У главном програму инстацира се 100 `Car` модула и сви се повезују преко исте семафор променљуве која се зове `park`. Семафор може да броји од 50 наниже. То значи да се у паркингу може налазити до 50 кола.

У свом процесу, модул `Car` чека случајан интервал времена и затим покушава да приступи паркинг месту, помоћу методе `wait` класе `sc_semaphore`. Уколико је бројач семафора позитиван, кола ће добити место, и бројач ће се смањити. У супротном, процес ће чекати да бројач постане позитиван. Процеси који прођу методу `wait`, штампају поруку, чекају случајан интервал времена и потом методом `post` повећавају бројач семафора.

1.7.3 sc_fifo

Један од најчешће коришћених канала у архитектурном моделовању је First-in First-out (FIFO) ред. У SystemC језику класа `sc_fifo` имплементира FIFO ред. FIFO су уобичајене структуре, намењене да

контролишу ток података и једне су од најједноставнијих структура за управљање.

Најважније методе за рад са модулом sc_fifo су:

```
void write(const T&);
bool nb_write(const T&);
void read(T&);
bool nb_read(T&);
int num_available();
int num_free();
```

Метода write уписује вредности прослеђене као аргумент у FIFO. Ако је FIFO пун, тада ова метода чека док се не ослободи место у реду. Метода nb_write идентична је методи write, једина разлика је да када је FIFO пун ова метода не чека да се ослобиди мето, већ враћа вредност false.

Метода read чита вредност из FIFO реда у промењиву која се проследи као параметар. Ако је FIFO празан, тада ова нетода чека да се нека вредност упише у ред. Метода nb_read идентична је методи read, осим када је FIFO празан. Тада, ова метода не чека да се нешто упише у ред, већ враћа вредност false.

Метода num_available враћа број података у FIFO реду, док метода num_free враћа број слободних места у FIFO реду.

Једноставан пример за употребу FIFO реда био би случај у коме два процеса уписују податке у FIFO, док их трећи процес чита (листинг 1.12). Модул FifoExample садржи једну променљиву типа sc_fifo и три процеса.

Листинг 1.12: Илустрација FIFO реда

```
#include <systemc>
#include <iostream>

using namespace sc_core;

SC_MODULE(FifoExample)
{
public:
	SC_HAS_PROCESS(FifoExample);

	FifoExample(sc_module_name, int);

protected:
	sc_fifo<int> fifo;

	void source1();
	void source2();
	void drain();
};

FifoExample::FifoExample(sc_module_name name, int size) : sc_module(name), fifo(size)
{
	SC_THREAD(source1);
	SC_THREAD(source2);
	SC_METHOD(drain);
```

```
}

void FifoExample::source1()
{
        int s1 = 100;
        while(1)
        {
                wait(rand()%3+3, SC_NS);
                fifo.write(++s1);
                std::cout
                        << "@" << sc_time_stamp() << " source 1 write "
                        << s1 << ".\n";
        }
}

void FifoExample::source2()
{
        int s2 = 300;
        while(1)
        {
                if(fifo.nb_write(s2))
                {
                        std::cout
                                << "@" << sc_time_stamp() << " source 2 write "
                                << s2 << ".\n";
                }
                else
                {
                        std::cout
                                << "@" << sc_time_stamp() << " fail write.\n";
                }
                wait(rand()%3+3, SC_NS);
        }
}


void FifoExample::drain()
{
        int d;
        for (int i = 0; i < 5; ++i)
                // Blocking read is not allowed in
                // SC_METHOD
                if(fifo.nb_read(d))
                {
                        std::cout
                                << "@" << sc_time_stamp() << " read "
                                << d << " data in fifo " << fifo.num_available() << std::endl;
                }
                else
                {
                        std::cout
                                << "@" << sc_time_stamp() << " fail write.\n";
                        break;
                }

        next_trigger(20, SC_NS);
}

int sc_main(int argc, char* argv[])
{
        const int FIFO_SIZE = 20;
        FifoExample uut("UUT", FIFO_SIZE);

        sc_start(200, SC_NS);

    return 0;
}
```

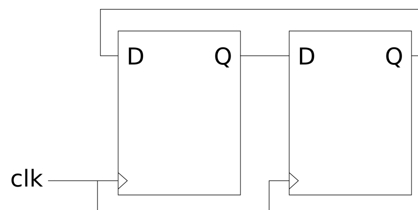
Процес `source1` пише податке у FIFO коришћењем методе `sc_fifo::write`. Овај процес чека случајан временски интервал, потом уписује податке у FIFO ред и на крају исписује поруку. Процес `source2` функционише на сличан начин, али је разлика што се користи метода `sc_fifo::nb_write`. Уколико процес не успе да упише податке у ред, штампа поруку о грешци.

Процес drain имплементиран је SC_METHOD процесом. Процес покушава да прочита вредности из FIFO реда 5 пута коришћењем методе sc_fifo::nb_read. Уколико се подаци успешно прочитају, штампа се порука о успеху. У супротном, престаје се са даљим читањем и исписује се порука о грешци. Процес потом чека 20 ns и понавља све исто.

У главном програму, инстанциран је модул FifoExample. Дубина FIFO реда прослеђена је као параметар модулу. Након инстанционирања, стартована је симулација у трајању од 200 ns.

1.7.4 sc_signal

Сигнали су канали који се углавном користе за моделовање хардверских елемената система. Да би размотрили проблем синхронизације, приликом моделовања хардвера, узмимо за пример везу два флип-флопа која размењују вредност на свакој ивици клока (слика 1.19).



Слика 1.19: Флип-флопови

Уколико флопови имају различиту почетне вредности, они ће их размењивати на свакој ивици и на њиховим излазима добиће се вредност која осцилује.

Ако би се излази флоопва моделовале обичним променљивим, тада би било врло компликовано добити одговарајуће понашање (листинг 1.13). Класа TwoFlops моделује излазе флопова са променљивим.

Листинг 1.13: Флип-флопови моделовани променљивим

```
#include <systemc>
#include <iostream>

using namespace sc_core;
using namespace std;

SC_MODULE(TwoFlops)
{
public:
	SC_HAS_PROCESS(TwoFlops);

	TwoFlops(sc_module_name);

	void print();

protected:
	bool v0, v1;
```

```
        void change();
};

TwoFlops::TwoFlops(sc_module_name name) : sc_module(name)
{
        SC_THREAD(change);
        v0 = false;
        v1 = true;
}

void TwoFlops::change()
{
        while(1)
        {
                wait(5, SC_NS);
                v0 = v1;
                v1 = v0;
        }
}

void TwoFlops::print()
{
        cout
                << "v0 = " << v0 << endl
                << "v1 = " << v1 << endl;
}

int sc_main(int argc, char* argv[])
{
        TwoFlops uut("UUT");

        sc_start(200, SC_NS);

        uut.print();

    return 0;
}
```

На крају симулације овог једноставног модела, флопови би имали идентичне вредност. Понашање обичних променљивих не одговара понашању правих флопова. Уз додатно усложњавање модела, могуће је добити исправно понањашње система. Но ово је једноставна ситуација, а модел коришћењем променљивих био би компликован.

Да би се омогућило једноставно моделовање хардверских елемената сустема у секвенцијалном језику, као што је C++, у кернел SystemC симулатора убачена је фаза ажурирања. Специјални канали, који се називају сигнали, користе фазу ажурирања као место синхронизације података. Да би се ово постигло, сваки сигнални канал, садржи две локације за чување података: тренутну вредност и нову вредност.

Када процес упише вредност у сигнални канал, податак се сачува у нову вредност, док тренутна вредност остаје непромењена. Потом процес позива `request_update` метод да би кернел симулатора позвао методу `update` одговарајућег канала у фази ажурирања.

Стога за моделовање хардверских елемената користе се сигнали. Канал сигнал је у SystemC језику имплементиран класом `sc_signal`. Најважније методе ове класе су:

```
void write(const T&);
void read(T&);
virtual const sc_event& default_event() const
```

Метода `write` служи за упис вредности у канал, док метода `read` омогућава читање вредности из канала. За овај канал преклопљени су и многи оператори, као што је на пример `operator=`. Стога, употреба овог канала, може бити једноставна колико и употреба обичне променљиве. Ипак препоручљиво је користити `read` и `write` методе, да не би дошло до мешања сигнала и променљивих, што може проузроковати грешке које се тешко проналазе.

Метода `default_event` омогућава да се добије догађај који шаље обавештења када год се промени вредност на сигналу. Ова метода је посебно занимљива, пошто омогућава да се сигнли шаљу на `sc_sensitive` објекат. Ова могућност се користи приликом моделовања комбинационих делова система приликом развоја RTL модела.

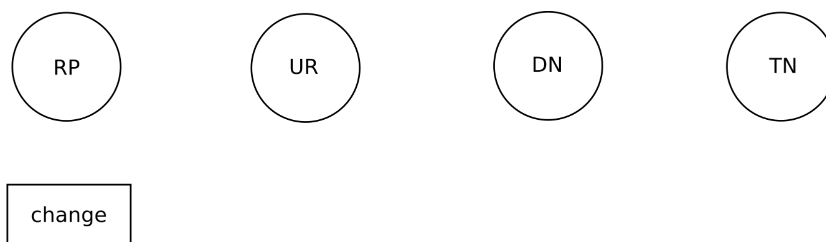
У случају када се излази флопова моделују сигналима, врло једноставно добија се коректно понашање система (листинг 1.14). Модул `TwoFlops` у овом случају излазе флопова моделује сигналима.

Листинг 1.14: Флип-флопови моделовани сигналима

```
#include <systemc>
#include <iostream>

using namespace sc_core;
using namespace std;

SC_MODULE(TwoFlops)
{
public:
        SC_HAS_PROCESS(TwoFlops);

        TwoFlops(sc_module_name);

        void print();

protected:
        sc_signal<bool> s0, s1;

        void change();
};

TwoFlops::TwoFlops(sc_module_name name) : sc_module(name)
{
        SC_THREAD(change);
        s0 = false;
        s1 = true;
}

void TwoFlops::change()
{
        while(1)
        {
                wait(5, SC_NS);
                s0 = s1;
                //s0.write(s1.read()); // This is alternative
                s1 = s0;
                //s1.write(s0.read()); // This is alternative
        }
}

void TwoFlops::print()
{
        cout
                << "s0 = " << s0 << endl
                << "s1 = " << s1 << endl;
}

int sc_main(int argc, char* argv[])
{
```

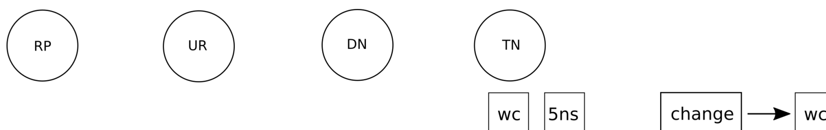
```
        TwoFlops uut("UUT");

        sc_start(200, SC_NS);

        uut.print();

    return 0;
}
```

На крају симулације, флопови ће имати различите вредности, као што и треба да буде. Следи кратко објашњење симулације у којој ће се видети шта се дешава у фази ажурирања. Након фазе иницијализације у скупу RP налазиће се процес change а тренутне вредности сигнала `s0` биће `false` а сигнала `s1` биће `true` (слика 1.20).



Слика 1.20: Стање пре прве евалуације

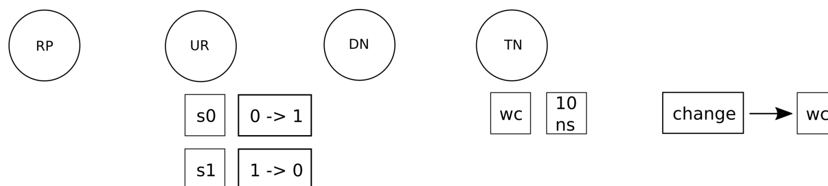
Процес change извршиће метод wait са временским аргументом и прећиће у стање чекања на обавештење од догађаја који ћемо звати `wc` (слика 1.21). Нако овога, фаза евалуације ће се завршити пошто је скуп RP празан, а у фазама ажурирања и делта кашњења неће се ништа десити. Након фазе делта кашњења, скуп RP ће бити празан па ће симулатор прећи у стање напредовања времена. У фази напредовања времена, догађај `wc` послаће своје обавештење, процес `change` пребациће се у скуп RP, и време ће напредовати 5 ns.



Слика 1.21: Стање након прве евалуације

Ситуација пре друге евалуације биће идентична ситуације пре прве евалуације, осим што ће процес `change` сада извршити други део кода. Овај пут, сигнали `s0` и `s1` измениће вредности. Иако се у измени вредности користи само оператор =, у имплементацији овог оператора позива се метода `request_update`, па ће оба канала, `s0` и `s1`, бити убачена у скуп UR. Канал `s0` за тренутну вредност има `false` а за нову вредност добиће тренутну вредност сигнала `s1` која је `true`. Сигнал `s1` садржи

тренутну вредност `true` а за нову вредност добиће тренутну вредност сигнала `s0`, која је `false`. Када се уради размена вредности сигнала, процес `change` позваће `wait` методу и завршиће своје извршавање. Скуп RP биће празан па ће кернел симулатора прећи у фазу ажурирања.



Слика 1.22: Стање након друге евалуације

Најзад, у фазу ажурирања скуп RU није празан (слика 1.22). У скупу се налазе два сигнална канала. За сваки од ових канала аутоматски се позива метода `update`. Ова метода пребацује нове вредности у тренутне. То значи да ће сигнал `s0` променити тренутну вредност са `false` на `true`, док ће `s1` променити вредности супротоно. Скуп RU остаће празан и кернел симулатора прећиће на фазу делта кашњења. Даље ће симулација тећи на већ описани начин. Метода `update` је заштићена и корисник не може да је позове, већ то може да уради само кернел симулатора.

Као што се могло видети у претходној илустрацији, фаза ажурирања је место синхронизације за сигналне канале. Као коначан резултат добија се понашање идентично понашању жица у дигиталним хардверским системима. Стога се за моделовање хардверских компонената, на нижим нивоима абстракције, углавном користе сигнални канали.

1.7.5 Специјализације и предефинисани канали

Специјализација шаблона догађа се када за одређене типове података постоји специјализована имплементација, другачија од генеричке. SystemC садржи неке специјализације шаблона за класу `sc_signal`.

Уколико се `sc_signal` користи са типовима података `bool` и `sc_logic` тада су на располагању додатне методе:

```
virtual const sc_event& posedge_event() const
virtual const sc_event& negedge_event() const
```

Ове методе враћају догађаје који шаљу обавештења, не на свакој промени сигнала, већ само уколико се деси одговарајућа промена. Догађај добијен са `posedge_event` шаље обавештења само када се

деси промена вредности сигнала на `true`, односно `SC_LOGIC_1`, док догађај добијен са `negedge_event` шаље обавештења приликом промена вредности на `false`, односно `SC_LOGIC_0`.

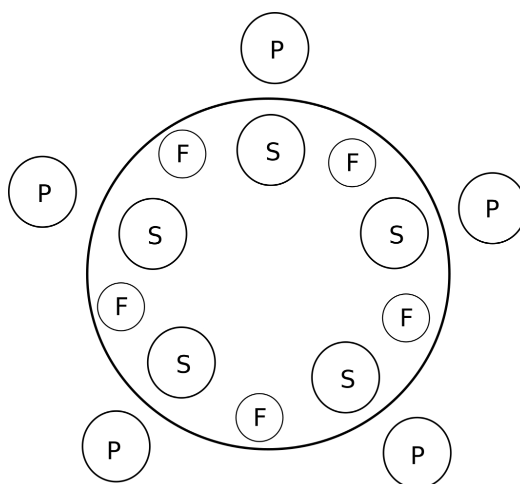
Приликом моделовања хардверских компоненти система, веома често јавља се потреба за синхронизационим каналом. Зато, језик SystemC садржи класу `sc_clock`, који наслеђује `sc_signal<bool>` и имплементира аутоматско стварање догађаја на задати интервал времена. Приликом конструкције модула `sc_clock` може се задати периода, фактор испуне, да ли ће клок стварати догађаје на растуће или опадајуће ивице и време почетка. Неки од конструктора су:

```
sc_clock()
sc_clock(const char *name_)
sc_clock(
    const char *name_,
    const sc_time &period_,
    double duty_cycle_=0.5,
    const sc_time &start_time_=SC_ZERO_TIME,
    bool posedge_first_=true)
```

1.7.6 Вежбе

1. Моделовати проблем пет филозофа (слика 1.23): пет тихих филозофа (слика - `P`) седи за округлим столом са тањирима шпагета (слика - `S`). Виљушке (слика - `F`) су постављаене између сваког пара суседних филозофа. Сваки филозоф мора размишљати и јести. Међутим, филозоф може јести шпагете само ако има и леву и десну виљушку. Сваку виљушку може држати само један филозоф. Након што појединачни филозоф поједе шпагете спушта обе виљушке како би виљушке постале доступне другима. Филозоф може узети леву или десну виљушку али не могу почети да једе пре него што добију обе виљушке.

   Филозофе моделовати као модуле. Направити само један модел филозова који ће се инстанцирати 5 пута у главном програму. Виљушке моделовати као мутексе. Када филозоф размишља а када покушава да једе моделовати по жељи. Поруку о тренутном стању филозова исписати сваке секунде симулације. Уколико се деси ситуација да сваки филозоф држи само једну виљушку, исписати одговарајућу поруку и завршити симулацију.



Слика 1.23: Тихи филозофи

2. Моделовати филтар задат релацијом 1.1. Комбинациони део филтра моделовати коришћењем `SC_METHOD` процеса. Све вредности филтра моделовати сигналима. Комбинациони процес треба да је осетљив на све сигнале. Секвенцијални део моделовати `SC_THREAD` процесом који се активира порукама унутрашњег сигнала `clk`. Трећи процес унутар филтра треба периодично да мења вредност сигнала `clk`. Уместо овог процеса може се користити и класа `sc_clock`. Вредности излаза добијене оваквим моделом упоредити са златним вредностима модела.