1.1 Симулација SystemC модела

SystemC је отворени стандард. Стога, SystemC модели могу да се симулаирају на свим симулаторима који имплементирају овај стандард. Симлуатори највећих фирми Electronic Design Automation (EDA) индустрије подржавају овај језик. Постоји и референтна имплементација стандарда која се назива и Open SystemC Initiative (OSCI) симулатор. За развој модела могу се користити и стандардна C++ развојна окружења, Integrated Development Environment (IDE), у које се укључује OSCI симулатор.

1.1.1 Инсталација OSCI SystemC симулатора

Језик SystemC може да се посматра и као обична C++ библиотека. На већини система ова библиотека није инсталирана, стога ју је потребно инсталирати. У даљем тексту, биће објашњено како се ова библиотека инсталира на UNIX оперативним системима, који имају Linux кернел, коришћењем GNU Compiler Collection (GCC) компајлера, односно преводиоца. Подразумеваће се и да је оперативни систем 64-битан. Поступак инсталације је сличан и на осталим оперативним системима као и одговарајућим преводиоцима.

Изворни код SystemC библиотеке, потребно је скинути са званичног сајта [1]. Изворни код је обично запакован у једну датотеку, коју је потребно распаковати. Потом је потребно променити радни директоријум на место где је датотека распакована и инсталирати библиотеку стандардном секвенцом наредби:

```
./configure --prefix=${SYSTEMC}
make
make install
```

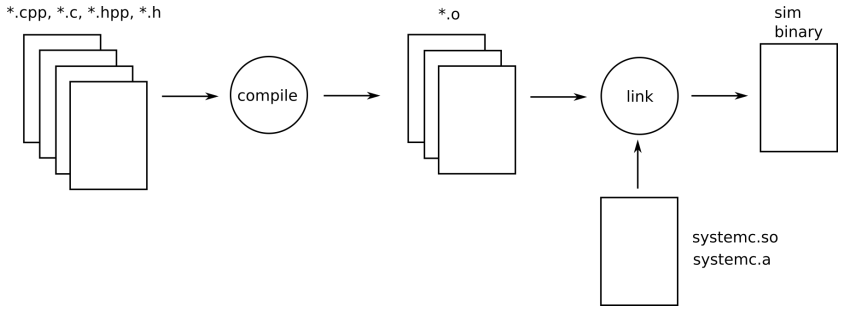
где је `$SYSTEMC` директоријум у који се библиотека инсталира. На пример, та вредност може бити `/opt/systemc`. У наставку текста подразумеваће се да је подешена променљива окружења, `$SYSTEMC`, која садржи име директоријума у ком је инсталирана библиотека.

Додатно, потребно је подесити променљиву окружења `LD_LIBRARY_PATH`, да садржи путању на којој се налазе датотека намењена динамичком повезивању. Ова променљива омогућава да се бинарне датотеке намењене симулацији динамички повежу са SystemC библиотеком. Датотеке за динамичко повезивање налазе се у директоријуму `lib-linux64` који се налази на месту где је SystemC библиотека инсталирана. Пример за подешавање ове променљиве је дат.

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}/lib-linux64
```

1.1.2 Компајлирање SystemC модела

Генерални поступак компајлирања SystemC одела је идентирачн компајлирању стандардних C++ програма. Изворне датотеке се, прво, компаљирају у објектне датотеке, помоћу компајлера (слика 1.1). Изворне датотеке најчешће имају екстензију `cpp` или `c`, док заглавља имају ектензију `hpp` и `h`. У случају GCC тока позива се `g++` компајлер. Као резултат компајлирања добијају се објектне датотеке које обићно имају екстензију `o`.



Слика 1.1: Компајлирање SystemC програма

Након компајлирања следи линковање, односно повезивање. У случају GCC тока, позива се `ld` линкер. Током повезивања као улази прослеђују се објектне датотеке добијене након компајлирања, као и SystemC библиотека. Могуће су две врсте повезивања: статичко и динамичко. Уколико се статички повезује библиотека, линкеру се у GCC току прослеђује датотка `systemc.a`, а у случају динамичког повезивања прослеђује се датотека `systemc.so`. Као резултат процеса добија се извршна бинарна датотека. Да би се симулација покренула довољно је покренути добијену бинарну датотеку.

Постоји већи број стандарда за C++ језик. SystemC библиотека може да ради са великим бројем њих. У приказаним моделима подразумваће се употреба C++ стандарда 2011. Стога, компајлеру ће морати да се проследи опција да се користи овај стандард.

1.1.3 Покретање SystemC OSCI симулације

Поступак компајлирања и симулирања SystemC модела биће демонстриран на једноставном програму (листинг 1.1). Програм само

исписује поруку али је за његово компајлирање неопходно да се уради повезивање са SystemC библиотеком.

Листинг 1.1: Једноставан SystemC програм

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

SC_MODULE (hello)
{
        SC_CTOR (hello)
        {
                std::cout << "Hello\n";
        }
};

int sc_main(int argc, char* argv[])
{
        hello h("hello");
        return(0);
}
```

Компајлирање и линковање могу да се изврше само једном командом када се користи GCC ток (листинг 1.2). За прављење бинарне симулационе датотеке, може се позвати само `g++` програм, који у свом извршавању позива `ld` програм, што је једноставније. Друга могућност је да се позивају оба програма одвојено, што даје додатну флексибилност. Било који приступ да се користи, потребно је са опцијом `-I` назначити компајлеру, у ком директоријуму се налазе SystemC заглавља. Заглавља се налазе у директоријуму `$SYSTEMC/include`. Додатно, потребно је назначити са `-L` директоријум где су смештене SystemC датотеке намењене динамичком и статикчом повезивању (`libsystemc.so` и `libsystemc.a`). Ове датотеке се налазе у директоријуму `$SYSTEMC/lib-linux64`.

Листинг 1.2: Компајлирање SystemC програма

```
all: hello_s hello_d

hello_s: hello.cpp
        g++ -std=c++11 -I ${SYSTEMC}/include -L${SYSTEMC}/lib-linux64 -l:libsystemc.a \
    -lpthread -o hello_s hello.cpp

hello_d: hello.cpp
        g++ -std=c++11 -I ${SYSTEMC}/include -L${SYSTEMC}/lib-linux64 -lsystemc \
    -o hello_d hello.cpp

.PHONY: clean
clean:
        rm hello_*
```

Уколико је потребно градити бинарну датотеку динамичким повезивањем, `hello_d`, потребно је повезати се са `libsystemc.so`. Ово се ради помоћу опције `-l`. Када се библиотека динамички повезује у GCC току, не наводи се префикс `lib` као ни екстензија `so`.

Уколико се гради датотека статичким повезивањем, `hello_s`, потребно је повезати се са `libsystemc.a`. Опет се користи опција `-l`, али се овај пут, наводи цело име датотеке раздвојено од опције симболом `:`. Додатно, неопходно је повезати се и са библиотеком pthre-

ads, која омогућава рад са нитима, пошто она није садржана у SystemC библиотеци. Ова библиотека је преинсталирана на већини UNIX система.

Да би се омогућила употреба C++ 2011 стандарда потребно је проследили опцију `-std=c++11` GCC компајлеру (листинг 1.2). Уколико се изостави ова опција неки модели неће моћи да се компајлирају, па стога ни симулирају.

Симулација се покреће једноставним позивом неке од добијених бинарних датотека, `hello_s` или `hello_d`. На пример:

```
./hello_d
```

Симулациона датотека `hello_s` је значајно веће величине од `hello_d`. Разлог за то је што се у њој налази цела SystemC библиотека. У случају када се позива `hello_d`, у току покретања програма односно симулације, SystemC библиотека се динамички учитава из датотеке `libsystemc.so`.

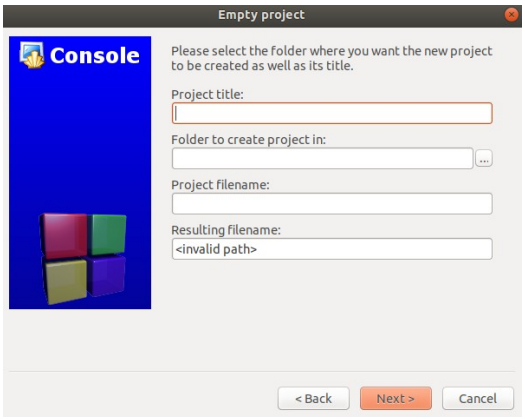
У наставку текста подразумеваће се употреба OSCI симулатора, осим уколико то није другачије наглашено.

1.1.4 C++ IDE

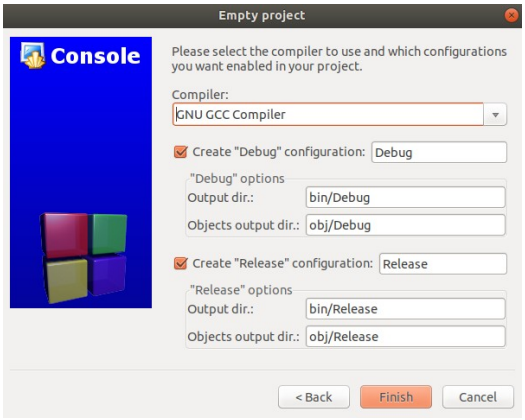
За развој SystemC модела могуће је користити и различита C++ развојна окружења, IDE, као што су на пример Code::Blocks или Eclipse CDT. Процедура је идентична као већ описана (слика 1.1), једино се подешавања за повезивање и укључивање заглавља морају подесити на прописани начин за одговарајући IDE. За приказ развоја SystemC модела, употребом развојних окружења, биће узет Code::Blocks.

Први корак за симулирање SystemC модела је прављење пројекта. То се постиже употребом `File -> New -> Project... -> Empty Project -> Go`. На овај начин добија се дијалог за подешавање опција празног пројекта (слика 1.2). Унутар дијалога потребно је дати име пројекту и одредити локацију на којој се смешта пројекат. Након тога је потребно урадити `Next` при чему се појављује дијалог за одабир компајлера (слика 1.3).

У оквиру прозора за одабир компајлера, у пољу `Compiler` потребно је одабрати жељени C++ преводиоц. У зависности од оперативног система на коме се развијају модели, као и од расположивих компајлера, ово поље може садржати разне подразумеване вредности. На оперативном систему Ubutnu, ово поље садржи GCC преводиоц. Остала подешавања повезана су са местом смештања објектних датотека и она могу да остану на подразумеваним вредностима. Притиском на акцију `Finish` завршава се прављење пројекта.

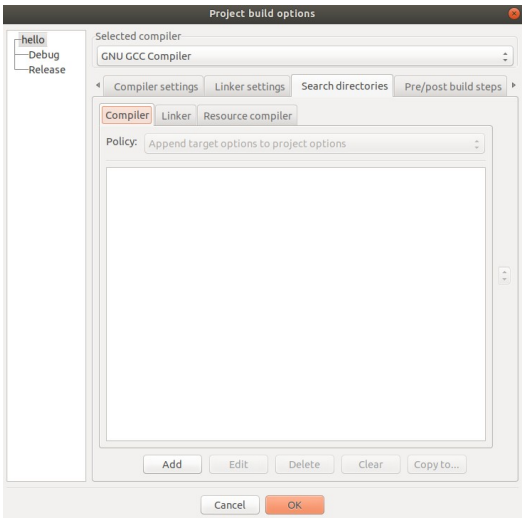


Слика 1.2: Дијалог празног пројекта



Слика 1.3: Одабир компајлера

Други корак је подешавање развојног окружења да може да користи SystemC библиотеку. Да би се дошло до ових подешавања потребно је урадити акцију `Project -> Build options...` након чега се појављује дијалог са многим подешавањима. Да би се омогућило компајлеру да пронађе SystemC заглавља потребно је одабрати картицу `Search directories`, па затим картицу `Compiler`, и на крају треба урадити акцију `Add` (слика 1.4). У дијалогу који се добије потребно је унети путању до SystemC заглавља. Подразумевано, направљени пројекат садржи два тока превођења: `Debug` и `Release`. `Debug` ток намењен је уклањану грешака током развоја. `Release` ток служи као коначни резултат при развоју. Бинарне датотеке добијене овим током, имају значајно боље перформајсе од `Debug` бинарних датотека, али су могућности уклањана грешака значајно мање. Приликом промене подешавања потребно је пазити који ток је тренутно активан. Затим је потребно подесити путању на којој се може наћи сама SystemC библитека. У картици `Search directories` се одабире картица `Linker` и акцијом `Add` добија се дијалог у коме се уноси путања до SystemC библиотеке.



Слика 1.4: Подешавања преводиоца - претрага путања

Трећи корак је додавање библиотеке у C++ пројекат. За то, потребно је одабрати картицу `Linker settings`, па у оквиру радног дела `Link libraries:` након акције `Add` појављује се дијалог у коме се уносе имена библиотека са којима пројекат треба да се повеже. За повезивање са SystemC библиотеком уноси се вредност `systemc`.

Четврти корак је додавање изворних датотека у пројекат. Уводни пример садржи само једну изворну датотеку `hello.cpp` (листинг 1.1). Акцијом `Project -> Add files...` добија се дијалог у коме се бирају изворне датотеке. Потребно је одабрати датотеку `hello.cpp`. Обично се додају само `*.cpp`, `*.cc` и `*.c` датотеке док, се заглавља прескачу.

Проласком кроз све наведене кораке, пројекат је спрема за градњу и покретање бинарне датотеке. Покретањем добијене бинарне датотеке, покреће се SystemC симулација. Изградња и порекретање бинарне датотеке могу се остварити акцијом `Build -> Build and run`. Након успешног завршетка симулације добија се прозор у коме је исписана порука `Hello`.

1.1.5 Комерцијални SystemC симулатору

Комерцијални симулатори највећих EDA компанија подржавају симулацију SystemC модела. Изворне датотеке моделе са укључују у симулацију модела на исти начин као и у случају Very High Speed Integrated Circuit Hardware Description Language (VHDL), Verilog или SystemVerilog модела. Ово, наравно, зависи од симулатора који се користи. У наставку биће приказано како се може покренути SystemC симулација, коришћењем симулатора Cadence® Xcelium™ Parallel Logic Simulation (Xcelium).

Комерцијални симулатори подржавају већи број токова, којима може да се покрене симулација. Ово омогућава развојним тимовима флексибилност приликом развоја модела. Најједноставнији начин да се покрене Xcelium симулатор је коришћењем команде `xmsc_run` (листинг 1.3). Овој команди се као аргумент прослеђују све извоне SystemC датотеке. Заглавља није потребно проследити.

Листинг 1.3: Покретање Xcelium симулатора

```
ncsc_run hello.cpp
```

Нако што се покрене команда, симулатор анализира изворне датотеке, елаборира их, оптимизује и на крају покреће саму симулацију. Све ове команде могу се покретати и свака засебно. Команда `xmsc_run` интегрише све наведене кораке у једну команду. Након успешно покренуте симулације, појављује се графички кориснички интерфејс - Graphical User Interface (GUI) симулатора, у оквиру кога могу да се раде све стандардне активности неопходне при развоју комплексних дигитални система. Могуће је приказивати вредности свих променљивих и сигнала, посматрати таласне облике, тражити грешке

помоћу интегрисаног система за помоћ при тражењу грешака и још много тога.

Лиценце за комерцијалне сумулаторе морају да се купе и то је главна мана ових симулатора у односу на слободно доступни OSCI симулатор. Но, комерцијални симулатор пружају разне погодности у односу на OSCI симулатор, које значајно олакшавају развој модела. Главна преднос ових симулатора, јесте могућност симулације модела писаних у раличитим језицима. OSCI симулатор може да симулира само SystemC моделе. Комерцијални симулатори могу да симулирају моделе писане у било ком од стандардних индустријских језика: Verilog, SystemVerilog, VHDL или SystemC. Оно што је још значајнија предност ових симулатора, и разлог због чега су незаменљиви при развоју у великим пројектима, јесте могућнос симулације модела писане у различитим језицима. У великим пројектима, готово увек, делови система моделовани су у једном од језика, док су неки други делови бити моделовани у неком другом језику. Стога је неопходан симулатор који подржава симулацију модела писаних у различитим језицима, а такви су само комерцијални симулатори, као што су: Xcelium, Mentor Graphics The Questa® Advanced Simulator (Questa) и Synopsys VCS® Functional Verification Solution (VCS).

У материјалу који следи, за моделе писане коришћењем само SystemC језика, подразумеваће се употреба OSCI симулатора. За моделе писане у раличитим језицима, подразумеваће се употреба неког од комерцијалних симулатора.

1.2 Одабрана поглавља C++ језика

Standard Template Library (STL) је стандардна C++ библиотека шаблонских класа, које имплементирају уобичајене контејнере, као што су низови, листе, стекови... Додатно, STL имплементира и стандардне алгоритме над овим контејнерима. Неки примери алгоритама су претрага и сортирање.

Идеја реализације ове библиотеке је да се оствари стандардни оквир над којим ће се имплементирати програми. Пре појаве STL-а, у многим пројектима, морали су да се имплеметирају контејнери, који би потом били коришћени само у оквиру тог једног пројекта. Тако су се појавиле многе имплементације истих контејнера, са различитим, најчешће некомпатибилним, начином коришћења. Да би се повећава употребна корисност контејнера направљена је STL библиотека.

У тексту који следи биће укратко приказане могућности STL библиотеке. За потпуно упознавање ове библиотеке, потребно је прочитати неку од многих књига на ову тему [2].

Контејнери садрже већи број вредности или објеката. У STL библиотеци имеплемтиран је већи број контејнера од којих су неки:

- vector
- list
- deque
- arrays
- stack
- map

STL библиотека се проширује и временом се додају нови контејнери. Илустрација начина коришћења контејнера биће приказана на класама `vector` и `list`.

1.2.1 STL `vector` контејнер

Класа `vector` имплементира динамички низ, који има могућност да мења своју величини када год се елемнт убацују или избацује из контејнера. Елементи су смештени један за другим у меморији, што омогућава брз насумичан приступ. Убацивање и избацивање елемената из класе `vector` је најефикасније на крају контејнера. Уколико се ове

операције спроводе на почетку контејнера, потребно је линеарно време за њихово завршавање.

Све врсте контејнера могу да се обилазе коришћењем итератора. Итератор је објекат који је као показивач, показује на елемент унутар контејнера. Итератори се користе слично као показивачи.

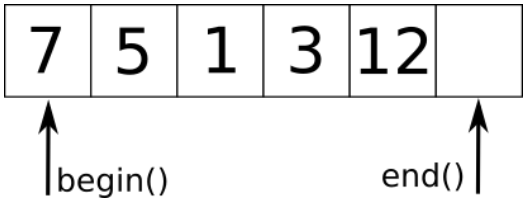
STL контејнери налазе се у простору имена `std` (листинг 1.4). Контејнери су шаблонске класе па их је потребно параметризовати другим типом података или класе. Дефиниција `std::vector<int> vec` дефинише променљиву `vec` која је `vector` контејнер, који се налази у простору имена `std` и који садржи целе бројеве типа `int`.

Листинг 1.4: Обилазак `vector` контејнера коришћењем итератора

```
#include <vector>
#include <iostream>

int main()
{
        std::vector<int> vec = {7, 5, 1, 3, 12};

        std::vector<int>::iterator it;

        for(it = vec.begin(); it != vec.end(); ++it)
                std::cout << *it << "␣";

        return 0;
}
```

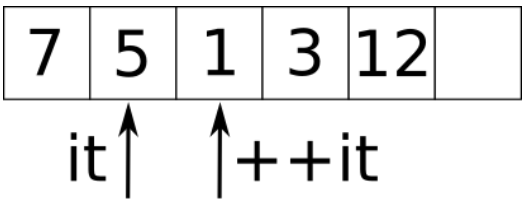
Да би се контејнер обилазио, једна од могућности је да се користе итератори (листинг 1.4). Дефиниција итератора увек садржи и контејнер по ком тај итератор може да итерира. Дефиниција `std::vector<int>::iterator it` дефинише променљиву `it` која је итератор, по контејнеру целих бројева `std::vector<int>`.

Методе контејнера `begin` и `end` као повратну вредност садрже итераторе на почетак контејнера, односно, на један елемент иза последњег елемента контејнера (слика 1.5). Јако је важно да метода `end` враћа итератор који заправо показује изван контејнера. Ово важи за све STL контејнере, не само за `vector`. Начин рада `end` методе, значајно поједностављује писање разних услова приликом обиласка STL контејнера. На пример, поређењем итератора са повратном вредношћу методе `end` може се знати да ли се дошло до краја контејнера, приликом секвенцијалног обиласка.



Слика 1.5: Итератори на почетак и крај контејнера

Итератори садрже многе преклопљене операторе, од којих је један оператор `++`. Овим оператором се итератор помера са тренутног елемента на наредни (слика 1.6). Сукцесивним позивом оператора `++` остварује се секвенцијални обилазак контејнера. Унарни оператор `*` омогућава приступ вредности унутар контејнера. Ово је исто као у случају показивача. Додатно, преклопљени су и оператори `==` и `!=` који омогућавају поређење да ли су два оператора идентична или су различита.



Слика 1.6: Секвенцијално померање итератора

Методе `begin` и `end` контејнера као и оператори `++`, `*` и `!=` омогућавају једноставан секвенцијални обилазак контејнера (листинг 1.4). Употребом `for` петље, итератор за секвенцијални обилазак (`it`) се може поставити на почетак контејнера `it = vec.begin()`. Затим се може тестирати да ли се дошло до краја контејнера `it != vec.end()`. Помоћи оператора `*` може се приступити елементу контејнера и урадити се са њим шта је потребно `*it`. На крају, могуће је померити итератор са тренутног елемента на наредни `++it`. Све ово се иделно уклапа у уобичајену `for` петљу.

Контејнер `vector` садржи још велики број метода које омогућавају једноставну манипулацију елемнетима. Неке од метода су:

- front() - Враћа вредност првог елемента контејнера.
- back() - Враћа вредност последњег елемента контејнера.
- оператор [] - Омогућава насумичан приступ елемнтима `vector` контејнера.
- push_front(el) - Додаје нови елемент `el` на почетак контејнера.
- push_back(el) - Додаје нови елемент `el` на крај `vector` контејнера.
- pop_front() - Уклања први елемент контејнера.
- pop_back() - Уклања последњи елемент контејнера.

- size() - Враћа број елемената унута контејнера.
- empty() - Тестира да ли је контејнер празан.
- begin() - Враћа итератор на први елемент контејнера.
- end() - Враћа итератор на теоријски елемент, који следи након последњег елемента контејнера.
- capacity() - Враћа величину простора која је тренутно заузета за употребу, изражену као број елемената који могу да стану у `vector`.
- resize(s) - Мења величину контејнера тако да садржи `s` елемената.

Наведене методе олакшавају употребу `vector` контејенра (листинг 1.5). Неке од наведених метода су универзалне за све контејенере, као сшо су `size` и `empty`. Неке методе су карактеристичне само за класу `vector`: `capacity`.

Листинг 1.5: Илустрација употребе неких метода `vector` контејнера

```
#include <vector>
#include <iostream>

int main()
{
        std::vector<int> vec;

        for(int i = 0; i != 5; ++i)
                vec.push_back(i);

        std::cout << "Size is " << vec.size() << "\n";
        std::cout << "Capacity is " << vec.capacity() << "\n";

        for(int i = 0; i != vec.size(); ++i)
                std::cout << vec[i] << " ";
        std::cout << std::endl;

        vec.resize(3);
        std::cout << "Size is " << vec.size() << "\n";
        std::cout << "Capacity is " << vec.capacity() << "\n";

        for(int i = 0; i != vec.size(); ++i)
                std::cout << vec[i] << " ";
        std::cout << std::endl;

        return 0;
}
```

1.2.2 STL `list` контејнер

Друга врста контејнера која се користи веома често су листе. Листе су секвенцијални контејнери који омогућавају смештање елемената у меморијске локације које се не налазе једна за другом. STL листе су имплементиране шаблонском класом `list`. Када се листе пореде са векторима, оне су значајно спорије за насумични приступ појединачним елементима, али је зато уметање елемената на произвољно место унутар

контејнера веома брзо. Класа `list` унутар STL библиотеке је двоструко спрегнута листа.

Обилазак `list` контејнера коришћењем итератора је идентичан обиласку вектора (листинг 1.6). Ова чињеница илуструје главну предност коришћења итератора за обилазак контејнера. Промена контејнера не узрокује никакву промену на већ писани део кода за обилазак. Једноставном употребом `typedef` могућа је промена контејенра.

Итератори омогућавају да се пишу функције и методе које су независне од тога који се контејнер користи. Захваљујући тој особине, омогућен је развој алгоритама који могу универзално да се користе (поглавље 1.2.3).

Листинг 1.6: Обилазак `list` контејнера коришћењем итератора

```
#include <list>
#include <iostream>

int main()
{
        std::list<int> vec = {7, 5, 1, 3, 12};

        std::list<int>::iterator it;

        for(it = vec.begin(); it != vec.end(); ++it)
                std::cout << *it << "␣";

        return 0;
}
```

Слично као и други STL контејнери, класа `list` садржи велики број метода.

- front() - Враћа вредност првог елемента контејнера.
- back() - Враћа вредност последњег елемента контејнера.
- push_front(el) - Додаје нови елемент `el` на почетак контејнера.
- push_back(el) - Додаје нови елемент `el` на крај контејнера.
- pop_front() - Уклања први елемент контејнера.
- pop_back() - Уклања последњи елемент контејнера.
- size() - Враћа број елемената унута контејнера.
- empty() - Тестира да ли је контејнер празан.
- begin() - Враћа итератор на први елемент контејнера.
- end() - Враћа итератор на теоријски елемент, који следи након последњег елемента контејнера.

- insert(iter, numel, el) - Умеће `numel` нових елемента у листу, са вредношћу `el`, испред позиције на коју показује итератор `iter`. Повратна вредност је итератор који показује на први од уметнух елемената.
- erase(iter) – Уклања елемнт на који показује итератор `iter`. Повратна вредност је итератор на први елемент након избрисаног.
- remove(val) – Уклања све елементе у листи који су једнаки `val`.

Излистане методе су само неке од метода класе `list`. Ове методе су стандардне, једноставне и препоручљиво их је користити у случајевима када је то могуће (листинг 1.7). Може се приметити да је велики број метода идентичан методама класе `vector`. Но, неке методе су изостављене, као што је на пример оператор за насумичан приступ `[]`. Насумичан приступ је веома спор у `list` контејнеру стога овај оператор није подржан.

Листинг 1.7: Илустрација употребе неких метода `list` контејнера

```
#include <list>
#include <iostream>

int main()
{
        std::list<float> reals;
        std::list<float>::iterator it, cur;

        for(int i = 0; i != 10; ++i)
                reals.push_back( 10.0 - (float)i);

        std::cout << "Size is " << reals.size() << "\n";

        int pos;
        for(pos = 0, it = reals.begin(); it != reals.end(); ++pos, ++it)
        {
                if (pos == 4) cur = it;
                std::cout << *it << " ";
        }
        std::cout << std::endl;

        std::cout << "Insert elements to list.\n";
        cur = reals.insert(cur, 2, 14.0);

        for(it = reals.begin(); it != reals.end(); ++it)
                std::cout << *it << " ";
        std::cout << std::endl;

        // Advance iterator 2 positions.
        advance(cur, 2);

        std::cout << "Erase element from list.\n";
        cur = reals.erase(cur);

        for(it = reals.begin(); it != reals.end(); ++it)
                std::cout << *it << " ";
        std::cout << std::endl;

        reals.remove(3);

        std::cout << "Remove element from list.\n";
        for(it = reals.begin(); it != reals.end(); ++it)
                std::cout << *it << " ";
        std::cout << std::endl;

        return 0;
}
```

1.2.3 STL алгоритми

STL библиотека у заглављу `algorithms` дефинише скуп алгоритама, намењених да се користе над колекцијом елемената. Ови алгоритми раде над контејнерима (`vector`, `list`...) и обезбеђују често коришћене операције над колекцијом елемената. Неки од тих алгоритама су сортирање и разне врсте претрага.

Сортирање је једна од основних операција које се примењују на колицији елемената. Елементи се распоређују на одговарајући начин, који може бити растући и опадајући. STL садржи функцију `sort()` која сортира елементе контејнера. Временска комплексност имплементације је $\mathcal{O}(n \log n)$. Прототип `sort` функције је: `sort(s, e, f)`, где је `s` итератор од кога почиње сортирање, док је `e` итератор који показује на елемент један иза последњег до ког је потребно сортирати елементе. Трећи параметар `f` је опцион и омгућује да се проселеди функција на основу које ће се сортирати елементи. Подразумевано, ова функција је оператор `<`.

Обично се сортирају цели контејнери, па се у `sort` функцију обично прослеђују итератори добијени матодама `begin` и `end` (листинг 1.8). У случају колекције целих бројева, подразумевано ће они бити сортирани у растућем редоследу. Ако је потребно сортирати бројеве у опадајућем редоследу, потребно је проследити одговарајућу функцију.

Листинг 1.8: Илустрација употребе `sort` функције

```
#include <iostream>
#include <vector>
#include <algorithm>

void print_array(std::vector<int>&);
bool my_func(int, int);

int main(int argc, char* argv[])
{
    int a[] = {7,9,11,4,10,2,15,12};
        // Create vector from ordinary C array.
    std::vector<int> wi (a, a + sizeof(a)/sizeof(int));

        std::cout << "Original vector.\n";
    print_array(wi);

    std::sort(wi.begin(), wi.end());
        std::cout << "Sorted vector.\n";
    print_array(wi);

    wi[5] = 100;
        std::cout << "Change one element.\n";
    print_array(wi);

    std::sort(wi.begin(), wi.end(), my_func);
        std::cout << "Custom function sorted vector.\n";
    print_array(wi);

    return 0;
}

void print_array(std::vector<int>& vi)
{
        for (std::vector<int>::iterator it = vi.begin(); it != vi.end(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
```

```
}

bool my_func(int a, int b)
{
    return a > b;
}
```

Уколико би алгоритми могли да раде само над уграђеним типовима података, не би били од јако велике користи. C++ садржи механизме који омогућавају да се, на једноставан начин, омогући коришћење алгоритама и на кориснички дефинисаним типовима података.

У случају `sort` функције за корисниче типове података, потребно је обезбедити оператор `<` који ради над тим подацима. Друга могућност је дефинисати одговарајућу функцију која ради над корисничким подацима и проселдити је `sort` функцији.

Нека класа `MyClass` садри два поља: цео број `id` и стринг `name` (листинг 1.9). У случају када се сортира колекција елемената ове класе, није јасно да ли они треба да се пореде по пољу `id` или `name`. Имплементација пријатељског оператора `<`, одређује да се у случају ове класе, сортирање ради по целој вредности `id`.

Листинг 1.9: Илустрација употребе `sort` функције и корисничких типова података

```
#include <iostream>
#include <vector>
#include <algorithm>

class MyClass
{
public:
        MyClass(int i, const char* name)
        {
                id = i;
                this-> name = std::string(name);
        }

        std::string getName() {return name;};
        int getID() {return id;};

        friend bool operator < (const MyClass& c1, const MyClass& c2)
        {
                return c1.id < c2.id;
        }

private:
        int id;
        std::string name;
};

void print_list(std::vector<MyClass>& lst)
{
    std::vector<MyClass>::iterator it;

    for ( it = lst.begin(); it != lst.end(); ++it)
        std::cout << it->getName() << ": " << it->getID() << "\n";
    std::cout << std::endl;
}

int main()
{
    std::vector<MyClass> lst;

    lst.push_back(MyClass(5, "Crome"));
    lst.push_back(MyClass(2, "Tome"));
    lst.push_back(MyClass(15, "Some"));
    lst.push_back(MyClass(0, "Last"));
    print_list(lst);
```

```
    std::sort(lst.begin(), lst.end());
    print_list(lst);
}
```

Имплементације алгоритама из заглавља `algorithms` могу да раде на већини STL колекција. Додатно, ови алгоритми често могу да ради и на обичним C низовима. Уколико алгоритми раде над обичним низовима, тада се итераторима сматрају обични показивачи.

Као додатни пример STL алгоритама, биће описана бинарна претрага. Бинарна претрага је веома брз алгоритам претраге који ради на сортираним колекцијама. У оквиру STL библиотеке, овај алгоритам имплементиран је `binary_search` функцијом. Прототип ове функције је: `binary_search(s, e, v)`, где је `s` итератор од кога почиње претрага, док је `e` итератор са којим се завршава претрага (једно место иза последњег). Параметар `v` је вредност која се тражи у колекцији. Повратна вредност је `true` уколико је вредност пронађена, док је у супротном `false`.

Рецимо да се низ који претражујемо зове `a` (листинг 1.10). Низ `a` је обичан C низ. Пре коришћења функције `binary_search` овај низ је потребно сортирати функцијом `sort`. Након сортирања, може се проверити да ли се одговарајући елемент налази у колекцији позивом функције `binary_search`.

Листинг 1.10: Илустрација употребе `binary_search` функције

```
#include <algorithm>
#include <iostream>

int main()
{
    int a[] = {4, 2, 9, 0, 6, 7, -1, -2, 5, 13, 8};
    int ar_size = sizeof(a) / sizeof(int);

        std::cout << "Unsorted array:";
        for (int i = 0; i != ar_size; ++i)
                std::cout << a[i] << " ";
        std::cout << std::endl;

        std::sort(a, a + ar_size);
        std::cout << "Sorted array:";
        for (int i = 0; i != ar_size; ++i)
                std::cout << a[i] << " ";
        std::cout << std::endl;

    if (std::binary_search(a, a + ar_size, 2))
                std::cout << "Element 2 found.\n";
    else
                std::cout << "Element 2 not found.\n";

    if (std::binary_search(a, a + ar_size, 10))
                std::cout << "Element 10 found.\n";
    else
                std::cout << "Element 10 not found.\n";

    return 0;
}
```

Претходно су приказане само основне могућности STL итератора, колекција и алгоритама. Могућности ове библиотеке су значајно веће. Додатно, сваки нови C++ стандард додаје нове могућности у ову библиотеку. Стога, потребно је пратити C++ стандарде и литературу

(на пример [2], [3]), да би се одржао корак са могућностима библиотеке као и C++ језика.

1. Спојити моделе 1.4 и 1.6 у један модел, кде се тип колекције може променити једним `typedefom`.

2. Моделовати студента класом `student`, која садржи поља `name`, `points`, `index`. Поље `name` је стринг, док су поља `points` и `index` цели бројеви. Написати програм који учитава студенте из текстуалне датотеке и потом их сортира. Први критеријум сортирања је по пољу `points`, а други по пољу `index`.

1.3 Типови података

SystemC подржава све стандардне типове података као и C++. Сви типови података који се могу дефинисати у језику C++, могу се исто дефинисати и у језику SystemC. Уграђени типови података за представљање бројева у језику C++ имају ширину такву да се ефикасно могу имплементирати на хост рачунару. Те ширине су најчешће умножци броја 8. Приликом моделовања хардверских система, користе се произвољне ширине за представљање бројева. Стога, уграђени типови података нису увек погодни за моделовање хардверских система.

SystemC има дефинисане типове података намењених моделовање нумеричких вредности са произвољним бројем бита у репрезентацији. Ови типови података олакшавају моделовање хардверских система.

1.3.1 Цели бројеви

За моделовање целих бројева ширине до 64 бита користе се типови података:

- sc_int<width>
- sc_uint<width>

Уколико је потребно моделовати целе бројеве чија је ширина већа од 64 бита на располагању су типови података:

- sc_bigint<width>
- sc_biguint<width>

Целобројни типови података који имају слово `u` у имену се користе за неозначене бројеве (од енглеске речи "unsigned"), док се остали користе за означене бројеве. За ове типове података преклопљени су сви стандардни оператори за целе бројеве је њихова употреба идентична уграђеним целобројиним типовима. Сви типови података су параметризовани са width параметром, који означава број бита репрезентације броја.

Због перформанси симулације најбоље је користити уграђене C++ целобројне типове. Уколико они нису одговарајући, ако је могуће, односно ако се користе ширине до 64 бита, боље је користити типове `sc_int` и `sc_uint` јер су бржи за симулацију од `sc_bigint` и `sc_biguint`.

1.3.2 Логичке вредности

За моделовање хардверских логичких вредности користе се типови података `sc_logic` и `sc_lv`. Ови типови података могу да представе вредности које се користе приликом моделовања хардвера:

- '1' - SC_LOGIC_1
- '0' - SC_LOGIC_0
- висока импеданса - SC_LOGIC_Z
- непозната вредност - SC_LOGIC_X

За једно-битне вредности се користи тип података `sc_logic` а за више-битне се користи `sc_lv` који је параметризован бројем бита.

1.3.3 Реални бројеви са фиксном тачком

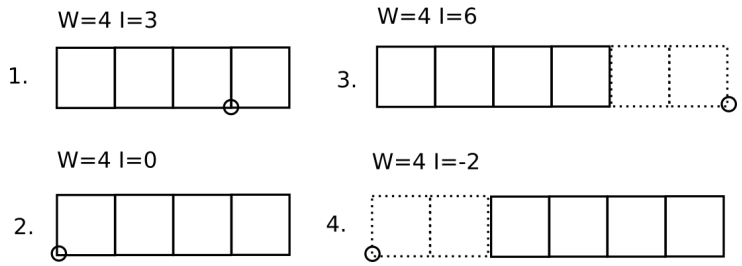
За представљаер реалних бројева у формату са фиксном тачком на располагању су следећи типови података:

- sc_fixed<W, I, Q, O, N>
- sc_ufixed<W, I, Q, O, N>
- sc_fixed_fast<W, I, Q, O, N>
- sc_ufixed_fast<W, I, Q, O, N>
- sc_fix_fast ctor(W, I, Q, O, N)
- sc_ufix_fast ctor(W, I, Q, O, N)
- sc_fixed_fast ctor(W, I, Q, O, N)
- sc_ufixed_fast ctor(W, I, Q, O, N)

Од излистаних типова података они који имају `fast` у имену се користе за предствљање бројева ширине до 53 бита. Уколико у имену постоји слово `u` ти типови података се користе за неозначене бројеве а остали за означене. Уколико тип података у имену има `ed` фиксни формат тог типа је статички подешен и не може се мењати током симулације. У супротном формат се може мењати током симулације и параметри се прслеђују приликом иницијализације података, као

аргументи у конструктору, а постоје и одговарајуће методе за мењање формата.

Параметар `W` одређује ширину броја. Параметар `I` одређује позицију фиксне тачке. Ова два параметра немају подразумеване вредности и морају им се додлити вредност пре коришћења.



Слика 1.7: Позиција фиксне тачке

На слици 1.7 су илустроване неке вредности параметара `W` и `I`. Ако разматрамо неозначене бројеве, тада се следећи бројеви могу представити:

- случај 1: од 0 до 7.5 са кораком 0.5.
- случај 2: од 0 до 0.9375 са кораком 0.0625.
- случај 3: од 0 до 64 са кораком 4.
- случај 4: од 0 до 0.234375 са кораком 0.015625.

Параметар `Q` одређује начин квантизације. Могуће вредности овог параметра су:

- SC_RND - заокруживање
- SC_RND_ZERO - заокруживање ка нули
- SC_RND_MIN_INF - заокруживање ка минус бесконачности
- SC_RND_INF - заокруживање ка бесконачности
- SC_RND_CONV - конвергирајуће заокруживање
- SC_TRN - одсецање
- SC_TRN_ZERO - одсецање ка нули

Параметар `O` одређује шта се дешава када дође до прекорачења. Могуће вредности овог параметра и одговарајућа решења су:

- SC_SAT - сатурација
- SC_SAT_ZERO - сатурација ка нули
- SC_SAT_SYM - симетрична сатурација
- SC_WRAP - премотавање
- SC_WRAP_SYM - симетрично премотавање

Да би се ови типови података могли користити потребно је убацити `#define SC_INCLUDE_FX` пре укључивања SystemC заглавља.

1.3.4 Пример

Као илустрацију употребе типова података који олакшавају моделовање хардверских елементата система, узећемо пример једноставог Infinite Impulse Response (IIR) филтра. Филтер је специфициран релацијом улаз излаз (једначина 1.1).

$$y(n) + 0.5y(n-1) + 0.2y(n-2) = 1.2x(n) + 0.3x(n-1) + 0.2x(n-2) \quad (1.1)$$

Јединични одзив овог филтра у првих 10 тачака је: 1.2000e+00, -3.0000e-01, 1.1000e-01, 5.0000e-03, -2.4500e-02, 1.1250e-02, -7.2500e-04, -1.8875e-03, 1.0888e-03, -1.6688e-04.

Сви параметри филтра биће представљени у формату фиксне тачке. У целом делу бројева налазиће се 2 бита. Симулацијом функционалног модела биће одређен најмањи број бита у разломљеном делу бројева тако да се јединични одзив моделованог филтра не разликује од задатог одзива за више од 0.001 у првих 10 тачака.

На почетку главне функције симулације, `sc_main`, дефинисани су параметри филтра (листинг 1.11), у прецизности `double`: `a_orig` и `b_orig` коефицијенти као и почетни услови `x_orig` и `y_orig`. Дефинисан је и злани вектор за овај модел, `gold`. Потом су дефинисани коефицијенти у ограниченој прецизности `a` и `b`, као и почетне вредности стања филтра `x` и `y`. Променљива `sys` садржаће прорачунати одзив филтра.

Листинг 1.11: Функционални модел једноставног филтра

```
#define SC_INCLUDE_FX
#include <systemc>
#include <iostream>
#include <deque>
#include <vector>
#include <cmath>

typedef sc_dt::sc_fix_fast num_t;
typedef std::deque<num_t> array_t;
typedef std::vector<double> orig_array_t;

void copy2fix(array_t& dest, const orig_array_t& src, int W, int F)
{
        for (size_t i = 0; i != src.size(); ++i)
        {
                num_t d(W, F);
                d = src[i];
                if (d.overflow_flag())
                        std::cout << "Overflow in conversion.\n";
                dest.push_back(d);
        }
}

bool passCheck(const orig_array_t& gold, const orig_array_t& sys,
                        double delta)
{
        for (size_t i = 0; i != gold.size(); ++i)
        {
                if (std::abs(gold[i] - sys[i]) > delta)
                        return false;
        }
        return true;
}

int sc_main(int argc, char* argv[])
{
        orig_array_t a_orig = {0.5, 0.2};
        orig_array_t b_orig = {1.2, 0.3, 0.2};
        orig_array_t x_orig = {1, 0, 0};
        orig_array_t y_orig = {0, 0};

        orig_array_t gold = {
                1.2000e+00, -3.0000e-01, 1.1000e-01, 5.0000e-03,
                -2.4500e-02, 1.1250e-02, -7.2500e-04, -1.8875e-03,
                1.0888e-03, -1.6688e-04
        };
        orig_array_t sys;

        array_t a;
        array_t b;
        array_t x;
        array_t y;

        const double error_d = 1e-3;
        int W = 5;
        const int F = 2;

        bool pass = false;

        // Main loop
        do
        {
                std::cout <<
                        "Starting pass for number format " << F <<
                        "." << W-F << std::endl;

                // Convert numbers
                copy2fix(a, a_orig, W, F);
                copy2fix(b, b_orig, W, F);
                copy2fix(x, x_orig, W, F);
                copy2fix(y, y_orig, W, F);

                // Calculate output
                for (size_t i = 0; i < 10; ++i)
                {
                        num_t sum(W, F);
                        sum = 0;

                        for (size_t i = 0; i != b_orig.size(); ++i)
                                sum += b[i] * x[i];
                        for (size_t i = 0; i != a_orig.size(); ++i)
                                sum -= a[i] * y[i];
```

```
                x.pop_back();
                x.push_front(0);
                y.pop_back();
                y.push_front(sum);

                sys.push_back(sum.to_double());
                std::cout << sum << " ";
            }
            std::cout << std::endl;

            // Check if error is small enough
            pass = passCheck(gold, sys, error_d);
            W++;
            sys.clear();
            a.clear();
            b.clear();
            x.clear();
            y.clear();
        } while(pass == false);

        std::cout << std::endl;

        return 0;
}
```

Главна петља симулације, која је `do-while` типа, састоји се из три дела:

- Конверзија бројева
- Рачун одзива
- Провера грешке

На почетку петље конвертују се бројеви. Функција `copy2fix` служи да конвертује бројеве из оригиналног формата вредности у покретном зарезу у вредности у репрезентацији са фиксном тачком. Параметри `W` и `F` одређују формат у фиксној тачки. Приликом сваког додатног пролаза кроз петљу параметар `W` се инкрементује, чиме се повећава прецизност бројева.

Потом се помоћу `for` петље прорачунва одзив филтра са параметрима у репрезентацији са фиксном тачком. Унутар променљиве `sum` смешта се вредност за тренутни одбирак одзива. Пошто су оператори за множење, сабирање и одузимање преклопљени за SystemC бројеве, прорачун изгледа идентично као да се ради са уграђеним C++ типовима података. Затим се ажурирају промнљиве стања и на крају се тренутна вредност `sum` додаје на рачунати одзив `sys`.

На крају главне петље проверава се грешка прорачунатог одзива у односу на златни одзив. Провера је имплементирана помоћу функције `passCheck`. Ова функција прима као параметре оба одзива и задату грешку и проверава да ли се нека два одбирка одзива разликују за више од задатке грешке. Уколико се разликују повећава се прецизност и ради се нова итерација. У супротном се главна петља прекида.

1.3.5 Вежбе

1. Написати функционални модел система који рачуна Фибоначијеву функцију (једначина 1.2).

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases} \tag{1.2}$$

   Функционални модел треба да је имплементиран целим бројевима ограничене ширине. Модел треба да може да рачуна вредности за све улазе мање од 64. Помоћу модела одредити минималан број бита за представљање бројева да би модел рачунао без грешке.

2. Написати два функционална модела система која раунају синус функцију на основу Тејлоровог развоја (једначина 1.3).

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \tag{1.3}$$

   Први модел треба да рачуна са бројевима у покретном зарезу, `C++ double` тип. На основу овог модела потребно је одредити број чинилаца у Тејлоровом развоју, неопходниг да се добије грешка мања од $1E^{-}5$.

   Други модел треба да рачуна са бројевима у фиксном зарезу. На основу овом модела потребно је одредити број бита у целом и разломљеном делу, тако да грешка овог модела буде мања од $1E^{-}5$ у односу на модел коју рачуна са покретном тачом.