Specifikace funkce modulů

• definicí předepsaných virtuálních funkcí v jazyku C++ ,

Složené moduly:

• reprezentovány objekty podtřídy cCompoudModule (odvozené od třídy cModule),

• nemají vlastní chování, zajišťují pouze distribuci zpráv,

• uživatel může specifikovat funkce:

virtual void cCompoudModule :: initialize ( ),

virtual void cCompoudModule :: finish ( ), // statistiky ,

Jednoduché moduly:

• reprezentovány objekty podtřídy cSimpleModule (odvozené od třídy cModule),

• uživatel může specifikovat funkce:

virtual void cSimpleModule :: initialize ( ), // inicializace

virtual void cSimpleModule :: handleMessage ( ), // příjem zprávy

virtual void cSimpleModule :: activate ( ), // popis procesu

virtual void cSimpleModule :: finish ( ), // výstup statistik

• orientace na události nebo na procesy:

– modul orientovaný na události nemůže volat activate ( ),

– modul s activate() nemůže volat handleMessage ( )

Algoritmus simulace

- **inicializace celého systému**:
  - postupným voláním koprogramů activate ( ) (u modulů orientovaných na procesy) nebo funkcí initialize ( ) (u modulů orientovaných na události),
  - funkce initialize ( ) vnitřního modulu je volána až po provedení této funkce u nadřazeného modulu,
- **algoritmus simulace**:

```
while ( ! konec simulace )
    {
        • vyjmi 1. zprávu (událost) z FEL ( objekt třídy odvozené
                                                od cMessage ),
        • simtime_t  t = hodnota času právě vyjmuté události,
        • nastav pointer m na vlastníka události:
        • if m odkazuje na složený objekt pak distribuce zprávy
          else
          { if ( m odkazuje na událostně orientovaný objekt
                                proveď m->handleMessage (  )
              else předej řízení na koprogram m->activate ( ); }
    }
```

- **ukončení simulace**:
  - if (řádný konec simulace) volání funkcí finish ( ) jednotlivých modulů počínaje jednoduchými moduly;
  - funkce finish ( ) složeného modulu je volána až po provedení této funkce u všech dílčích modulů (shromažďování statistik zdola)

Specifikace funkce jednoduchých modulů

Příklad: předpokládejme existenci modulu v NED:

simple M1

parameters:……..;

gates:……..;

endmodule

- pro všechny jednoduché moduly specifikované v NED je třeba definovat jejich vnitřní funkce v jazyku C++:

1) deklarovat hlavičky příslušné podtřídy třídy cSimpleModule včetně deklarace konstruktoru nebo volání makra Module_Class_Members pro jeho expanzi, syntax volání makra:

Module_Class_Members

(< identifikátor modulu z NED >, cSimpleModule,

< velikost zásobníku >)

< velikost zásobníku > : = 16K až 32K pro activate( ),

0 pro handleMessage ( ),

příklady:

varianta s makrem:

class M1 : public cSimpleModule

{ …..; Module_Class_Members (M1 , cSimpleModule, 0) } ;

varianta bez makra:

class M1 : public cSimpleModule

{ …..; public : M1( ) : cSimpleModule (0) {……. } ;

Specifikace funkce jednoduchých modulů

Poznámka: pokud bychom chtěli inicializovat nějaké atributy dané podtřídy tak je třeba použít konstruktor,

2) zajistit návaznost na rozhraní modulu v NED:

– voláním makra Define_Module (< identifikátor modulu z NED >),

– pro složené moduly je toto makro voláno automaticky

příklad:

Define_Module (M1); // vložit do souboru xxxx .cpp

3) specifikovat virtuální funkce třídy cSimpleModule.

```
void M1 :: initialize ( )
            {……………} ;
void M1 :: handleMessage ( )      // pouze pro událostně
                                  // orientované moduly
            {……………} ;
void M1 :: activity ( )           // pouze pro procesově
                                  // orientované moduly
            {……………} ;
void M1 :: finish ( )
            {……………} ;
```

Struktura modelu v systému OMNET++

1) Specifikace struktury sitě:

- vstupy překladače NED (1 nebo více souborů . ned):
  Příklad:
  - obsah souboru xxxxxx . **ned**: // rozhraní typu SM1
    simple SM1……..**endsimple**

  - obsah souboru yyyyyy. **ned**:
    // rozhraní SM2+celkové propojení
    import "xxxxxx . ned"
    simple SM2……..**endsimple** // rozhraní SM2
    module M // složený modul
    submodules: s1: SM1…….; s2: SM2……;
    connectiones:……; // propojení instancí
    endmodule
    // pojmenování instance modelu
    network sit: M parameters: …….**endnetwork**

2) Specifikace nestandardních atributů zpráv:

- vstup překladače zpráv
  - obsah souboru zzzzzz. **msg**
    message zprava1 {……..};
    message zprava2 {……..};

Struktura modelu v systému OMNET++

3) Specifikace chování funkčních modulů:

- obsah souboru ffffffff. **cpp:**

```
# include “zzzzzz_m.h“                // representace zpráv vC++
# include <omnetpp.h>
# define STACKSIZE 16384        // 16 k
 // následují hlavičky jednoduchých modulů
class SM1: public cSimpleModule
{ Module_Class_Members (SM1,cSimpleModule, STACKSIZE);
  virtual void activity();……..} ;

class SM2: public cSimpleModule
{ Module_Class_Members ( SM1,cSimpleModule, 0 );
   virtual void handleMessage();……..} ;;
Define_Module (SM1);
Define_Module (SM2);

// následuje definice funkcí specifikovaných tříd:
void SM1:: activity ()              { ……}
void SM2:: handleMessage () { ……}
```

4) Specifikace konfiguračního souboru:

- obsah souboru kkkkkk. **ini**:
  - volba modelu a parametrů pro jednotlivé simulační běhy, atd (viz později.)

Struktura modelu v systému OMNET++

Nejjednodušší možný model:

- obsah souboru priklad. **ned**:

```
simple SM                    // žádné rozhraní
endsimple
// žádná struktura
network s: SM                // definice sítě
endnetwork
```

- obsah souboru priklad. **cpp**:

```
# include <omnetpp.h>
class SM: public  cSimpleModule
{  virtual void  activity();     // jediný proces
    Module_Class_Members ( SM, cSimpleModule, 16384 )   };

 Define_Module( SM );
virtual void SM :: activity ()
 { ……. }        // specifikace procesu
```

- obsah souboru priklad. **ini**:

```
[ General ]
network = s            // volba modelu
[ Cmdenv ]             // textové rozhraní
[ Tkenv ]              // grafické rozhraní
[ Run1 ]               // 1. simulační běh
```

Vlastnosti virtuálních funkcí

– virtual void cSimpleModule :: initialize ( ),

- spuštění na začátku simulace,
- může plánovat volání funkce handleMessage ( ),

– virtual void cSimpleModule :: handleMessage
(cMessage * msg)

- volána automaticky při přijetí jakékoliv zprávy příslušným modulem; parametr msg vrací odkaz na přijatou zprávu,

- při specifikaci odezvy modulu lze v této funkci volat již definované funkce:
  - send ( ) ….odeslání zprávy do výstupního portu,
  - scheduleAt ( )….naplánování vlastní zprávy (tj. zprávy sobě),
  - cancelEvent ( ) …zrušení zprávy odeslané pomocí scheduleAt ( ),

- nelze volat funkce receive ( ) a wait ( ),

Vlastnosti virtuálních funkcí

- virtual void cSimpleModule :: activity ( ),
  - tato funkce definuje proces (koprogram), který múže být dobrovolně suspendován,
  - při specifikaci odezvy modulu lze v této funkci volat již definované funkce:
    - send ( ) …pro odeslání zprávy do výstupního portu,
    - scheduleAt ( )…naplánování vlastní zprávy,
    - cancelEvent ( ) … pro zrušení zprávy odeslané pomocí scheduleAt ( ),
    - receive ( )… suspenduje provádění funkce activity ( ); toto provádění bude pokračovat po přijetí zprávy v daném modulu,
    - wait ( )…suspenduje provádění funkce activity( ) na daný časový interval,
- virtual void cSimpleModule :: finish ( ),
  - volání: po ukončení simulace
  - účel: záznam statistických informací

Komunikační funkce modulu

- zprávy: objekty třídy cMessage nebo z ní odvozených podtříd (odkazované pomocí pointrů): slouží ke komunikaci modulů,
- vytváření: new, rušení: delete

Funkce pro odesílání zpráv: prostřednictvím výstupních portů:

```
int cSimpleModule :: send
      ( cMessage * msg,  const char * jméno portu, int index = 0 );

int cSimpleModule :: send ( cMessage * msg,   int  ident );

int cSimpleModule :: send ( cMessage * msg,   cGate *gate );
```

význam: msg……...ukazatel na zprávu,
jmeno……jméno portu, případně vektoru portů,
index……určuje pozici portu ve vektoru,
ident…….identifikace portu (rychlejší)
gate……..ukazatel na port,

Poznámka: zpráva může být odeslána pouze vlastníkem zprávy (kontrolováno v send); po odeslání nebo naplánování zprávy není modul již jejím vlastníkem => nelze odeslat znovu

- broadcasting: posílání kopií téže zprávy v cyklu,
- retransmission: odeslání duplikátu

Komunikační funkce modulu

Zpožděné odeslání zprávy: prostřednitvím výstupních portů

- odeslání v čase: simTime( ) + del

```
int cSimpleModule :: sendDelayed  ( cMessage* msg,
                        double del,  const char * jméno portu,
                        int index = 0 );
int cSimpleModule :: sendDelayed
                        ( cMessage* msg,  double del,  int  ident );

int cSimpleModule :: sendDelayed
                       ( cMessage* msg, double del, cGate*gate );
```

Přímé posílání zpráv: mimo vlastní výstupní porty

- na přijetí přímé zprávy reaguje modul standardním způsobem (stejné jako přes komunikační linku)
- zasílání zpráv do modulu s nepřipojeným vstupním portem

```
int cSimpleModule :: sendDirect  ( cMessage * msg, double del,
                        cModule* m,  const char * jméno portu,
                        int index = 0 );
int cSimpleModule :: sendDirect ( cMessage * msg,  double del,
                        cModule* m,  int  ident );
```

- zasílání zpráv do výstupního portu jiného modulu:

```
int cSimpleModule :: sendDirect ( cMessage * msg, double del,
                                  cGate *gate );
```

Funkce pro pozdržování procesu:

1) časové pozdržení procesu activity ( ):

void cSimpleModule :: wait (simtime_t del) :

- vnitřní implementace jako scheduleAt (simTime ( ) + del);
  receive ( ) ;
- přijetí jakékoliv zprávy v průběhu čekání způsobí chybu,
- použití : generátory časově oddělených zpráv

void cSimpleModule :: waitAndEnqueue
(simtime_t del , cQueue * fronta);

- zprávy přijaté během čekání (po dobu del) jsou zařazeny do fronty odkazované ukazatelem fronta

Příklad:

```
cQueue  fr ( " fronta " );
…….;
waitAndEnqueue ( del, & fr );        // pokračuj za dobu del
if ( ! fr . empty ( ) )  {…….}       // zpracuj přijaté zprávy
```

Funkce pro pozdržování procesu:

2) Podmíněné pozdržení procesu activity ( ):

(přijímání zpráv)

cMessage* cSimpleModule :: receive ( )

…suspenduje provádění funkce activity ( ) a čeká na přijetí zprávy, pak vrací pointer na tuto zprávu

cMessage* cSimpleModule :: receive (simtime_t time_out)

…suspendace jako u receive ( ), ale čeká na přijetí zprávy po dobu time_out; pokud vyprší time_out pak vrací NULL a výpočet pokračuje,

Příklad:

```
simtime_t  time_out = 3.0;
cMessage * msg = receive ( time_out ) ;
 if ( msg = NULL )
            { ……  }  // zpráva v dané době nebyla přijata
       else  { ……  }   // zpracování zprávy
```

Plánování vlastních zpráv

- potřeba měření času (generování časově závislých událostí: doba obsluhy, vyčerpání timeout, atd.)
- implementace pomocí zasílání zpráv „sobě“ (self -messages):
- naplánování vlastní zprávy:

  int cSimpleModule :: scheduleAt

  (simtime_t t , cMessage *msg);
- přijetí vlastní zprávy:

  virtual void cSimpleModule :: handleMessage (…) nebo

  cMessage* cSimpleModule :: receive ( )

- zrušení naplánované vlastní zprávy (pokud není rušená zpráva ve FEL, pak chyba) :

  cMessage* cSimpleModule :: cancelEvent

  (cMessage * msg);
- indikace vlastní zprávy

  a) porovnáním ukazatelů na odeslanou zprávu a na přijatou zprávu,

  Příklad: varianta pro procesově orientovaný popis

```
cMessage *msg = new cMessage ( );        // generování
scheduleAt ( simTime ( ) + 10.0,  msg );   // naplánování
cMessage * prijato = receive ( );        // čekání na příjem
if ( prijato == msg ) {  // vlastní zpráva   }
                  else {   // cizí zpráva       }
```

Indikace vlastních zpráv

b) pomocí funkce bool cMessage :: isSelfMessage ( )…….vrací
true jde-li o vlastní zprávu

Příklad: implementace časovače: vyvolání události za 10 sec

- varianta pro procesově orientovaný popis:

```
cMessage * zvonek = new Message ( " zvoneni " ) ;
scheduleAt ( simTime ( )  + 10.0 ) , zvonek );  // naplánování
                                                // vlastní zprávy
cMessage* prijato = receive ( );        // čekání na přijetí zprávy
 if ( prijato → isSelfMessage ( ) )
                          { ……// zvoní, časovač vyčerpán  }
                    else { cancelEvent ( zvonek) ;
                           ……; // zpracování došlé zprávy
                          }
```

- varianta pro událostně orientovaný modul:

```
void …… ::  handleMessage ( * msg );
    { if msg  → ( isSelfMessage ( ) )
                    {…….;    // zpracování vlastní zprávy ( zvoní )
                    }
                    else {
                          ……// zpracování cizí zprávy }
                         }
     ……..;
    }
```

c) existují i jiné možnosti (viz později)

Přístupy k parametrům modulu

Příklad: předpokládejme existenci modulu v NED:

simple M1

parameters: a, ……..; **gates**:……..;

endmodule

- parametry definované v jazyku NED:
  - uloženy (samostatně) v objektech třídy cPar (nosiče různých typů),
  - při výpočtu je potřeba získávat hodnoty parametrů, případně měnit jejich hodnoty,
- přístup k objektů třídy cPar:

  cPar& cModule :: par (const char* name)
  - vrací odkaz na nosič parametru s uvedeným jménem,
- funkce pro čtení hodnot parametrů:

  bool cPar :: boolValue()…. … vrací hodnotu jako typ bool

  long cPar :: longValue()…. … vrací hodnotu jako typ long

  double cPar :: doubleValue()

  const char* cPar :: stringValue()
- funkce pro zápis hodnot parametrů:

  cPar& cPar :: setBoolValue (bool b)

  cPar& cPar :: setLongValue (lond l)

  cPar& cPar :: setDoubleValue (double d)

  cPar& cPar :: setStringValue (const char*)

Přístupy k parametrům modulu

Příklady použití:

```
cPar& a  = par ( “a” );  // inicializace adresy nosiče parametru a
double a1 = a . doubleValue();              // získání hodnoty
double b1 = par ( “a“ ) . doubleValue ( );  //  jinak
a . setDoubleValue ( 3.14 );   // změna hodnoty v cPar
```

Jiné možnosti přístupu:

- využití přetížených operátorů pro přiřazení:

```
cPar& cPar::operator = (bool b)  // jako setBoolValue (bool b)
cPar& cPar:: operator = (long l) // jako setLongValue (long i)
cPar& cPar:: operator = (double d) // jako setDoubleValue ( ..)
cPar& cPar:: operator = (const char* s) // setStringValue ( ...)
```

- využití přetížených operátorů pro konverzi:

```
cPar::operator bool ( )          // jako boolValue()
cPar:: operator long ( )         // jako longValue()
cPar:: operator double ( )       // jako doubleValue()
cPar:: operator const char* ( )  // jako stringValue()
```

Varianty použití:

```
cPar&  a1 = par ( “a“ );    a1 = 1.189 ;  // přetížený oper. přiřazení
double a2 = ( double ) par ( “a“ );       // explictní konverze
double  a3;   a 3 = a1;          // operátor konverze
double a4 =  par ( “a“ );        // operátor konverze
```

Změny parametrů v průběhu výpočtu

Podpůrné funkce:

// následuje funkce pro specifikaci výzvy ke změně parametru

void cPar::setPrompt (const char * *s*) // specifikace výzvy

void cPar::setInput (bool b) // nastavení příznaku input

.. if b = true, resp. false pak funkce nastaví input flag parametru na 1, resp. na 0;

.. při přístupu k hodnotě parametru jehož input flag = 1 dojde k přerušení výpočtu a tisku nastavené výzvy; po zápisu nové hodnoty výpočet pokračuje,

Příklad:

```
cPar a ( “a” ) ;   // deklarace instance třídy cPar
a . setPrompt ( “ Nastav hodnotu a: “ ) ;   // uložení výzvy
a . setInput ( true);     // nastav input flag = 1
double b = ( double ) a;   // použití a => výstup  výzvy a čekání na
                           // novou hodnotu
```

Přesměrování hodnoty parametru do jiných objektů třídy cPar:

cPar& cPar::setRedirection (cPar * *par*)

…při změně hodnoty dané instance funkce zajistí uložení nové hodnoty i do instance určené parametrem par

Příklad:

```
cPar a1 ( “a1“ );  cPar  b1;
 a1 . setRedirection ( b1);   // pozor: nuluje hodnotu a1
 a1. setDoubleValue (2) ; zápis hodnoty 2 do a1 i b1
```

Ukončení simulace a určování času

- ukončování simulace:
  - v programu:
    - volání funkce endSimulation ( )
    - specifikace horní hranice simulačního nebo reálného času (v souboru . ini),
    - volání funkce void cModule :: error ( ),

      příklad: if (okno < 1) error (" velikost okna % d ", okno);
  - interaktivně z klávesnice:
    - tlačítko STOP

- zjišťování okamžitých hodnot simulačního času:
  - volání funkce simtime_t cSimpleModule :: simTime ( );

- jednotky simulačního času :
  - možno použít pouze při popisu struktury (NED),
  - implicitní jednotka: 1 s,
  - další možné jednotky: ns, us, ms, s, m, h, d,

    (ekvivalentní zápisy: 50 ,= 50s = 1 m)

Jednoduchý příklad vzájemné komunikace

Příklad: zjednodušení a úprava sitě ping_pong (viz demonstrační příklady systému OMNET++). Síť obsahuje 2 moduly m1 a m2, které si vyměňují zprávy :

- začne vysílat začíná modul m1,
- simulaci ukončí modul m2 po desátém vrácení zprávy.

- deklarace typů a specifikace výsledné struktury:

simple M // deklarace typu jednoduchého modulu
parameters: pocet : **numeric**; // celkový počet zpráv
gates : **in** : in; // 1 vstupní port
out: out; // 1 výstupní port
endsimple

module Celek // deklarace typu složeného modulu
submodules: // následuje deklarace dvou instancí
m1 : M; // m1…jméno instance
m2 : M;
connections: // následuje propojení instancí
m1. **out** → delay 100 ms → m2 . **in;**
m1. **in** ← delay 100 ms ← m2 . **out**;
endmodule
// deklarace instance pro reprezentaci výsledného modelu
network celek : Celek
endnetwork

Jednoduchý příklad vzájemné komunikace

varianta 1: zpráva je vrácena ihned po jejím obdržení

- popis typu modulu M:

```
class M : public cSimpleModule // odvození od základní třídy
{  int citac;                  // pro cítání odeslaných zpráv
   cMessage* msg ;      // odkaz na přijaté zprávy
   Module_Class_Members ( M, cSimpleModule, 0 );
   virtual void initialize ( ) ;
   virtual void handleMessage ( cMessage * msg );      } ;

Define_Module  ( M ) ;        // vazba na NED: typ M
void M :: initialize ( )      // spuštění na začátku simulace
{  citac = par ( " pocet " ) ;   // čtení hodnoty parametru z NED
   if  ( strcmp ( " m1", name ( ) ) == 0 )  // test jména modulu
         {    // následuje vytvoření prvé zprávy a její odeslání
           msg = new cMessage ( " micek " ) ;
           send  ( msg,  " out " );              }    // začíná m1
}
void M :: handleMessage ( cMessage * msg )
{    // reakce modulu na přijetí každé zprávy
    if  ( strcmp ( " m2", name ( ) ) == 0 )   // reakce modulu m2
              {   if ( citac == 0 ) { delete msg; endSimulation ( ); }
                          else   citac =  citac - 1 ;      }
    send  ( msg,  " out " );     // reakce obou modulů
}
```

Jednoduchý příklad vzájemné komunikace

varianta 2: zpráva je vrácena po zpracování trvajícím 2 sec

```
class M : public cSimpleModule
 {  int citac; cMessage *  vlastni, msg, info; // ukazatele zpráv
    Module_Class_Members ( M, cSimpleModule, 0 );
    virtual void initialize ( ) ;
    virtual void handleMessage ( cMessage * msg );      } ;

Define_Module  ( M ) ;                  // vazba na NED
void M :: initialize ( )        // spuštění na začátku simulace
{ vlastní = new cMessage ( " budik " ) ; //příprava vlastní zprávy
   citac = par ( " pocet " )  // čtení hodnoty parametru z NED
    if  ( strcmp ( " m1", name ( ) ) == 0 )  // test jména modulu
          {   msg = new cMessage ( " micek " ) ;
              send  ( msg,  " out " );                }    // začíná m1
void M :: handleMessage ( cMessage * msg )
 {   if ( msg != vlastni )        // rozlišení zpráv
          {  if  ( strcmp ( " m2", name ( ) ) == 0 )  // reakce m2
                 { if ( citac == 0 ) { delete msg; endSimulation ( ); }
                                        else  citac =  citac - 1 ;    }
             info = msg;            // paměť cizí přijaté zprávy
             scheduleAt ( simTime ( ) + 2.0, vlastní  );   // budík
          }
    else     send  ( info,  " out " ) ;   // vrácení zprávy
 }
```

Jednoduchý příklad vzájemné komunikace

varianta 3: funkce jako u varianty 2, ale s procesem activity ( ):

```
class M : public cSimpleModule
 {  int citac;                    // pro cítání odeslaných zpráv
    Module_Class_Members ( M, cSimpleModule, 8192 );  // 8K
    virtual void activity ( );            }

Define_Module  ( M ) ;
void M :: activity ( )    // procesově orientované chování modulu
{
  citac = par ( " pocet " ) // čtení hodnoty parametru z NED
  if  ( strcmp ( " m1", name ( ) ) == 0 )  // test jména modulu
              {    // následuje vytvoření zprávy a první odeslání
                cMessage * msg = new cMessage ( " micek " ) ;
                send  ( msg,  " out " ) ;
              }
  while ( 1 )       // následuje periodická funkce
     {  cMessage  * msg = receive ( );     // čeká na přijetí zprávy
        if  ( strcmp ( " m2", name ( ) ) == 0 )
                    { if ( citac == 0 ) { delete msg; endSimulation ( ); }
                                    else     citac =  citac - 1 ;   } // end if
        wait ( 2.0 );                       // doba zpracování zprávy
        send ( msg, "out" );                // vrácení zprávy
     }
}
```

Vlastnictví objektů

- mechanismus vlastnických vztahů:
  - objekty odvozené od třídy cObject mohou figurovat jako vlastníci jiných objektů nebo mohou být vlastněny,
  - vlastnictví je udržováno automaticky:
    - objekt „vlastník“ udržuje seznam všech jím vlastněných objektů, např. zpráva vlastní vloženou zprávu, fronta vlastní všechny dílčí zprávy, atd.
    - nově vygenerovaný objekt je vlastněn modulem, který jej vygeneroval,
    - vlastnictví zpráv se při komunikaci automaticky mění: po odeslání zprávy modul již není jejím vlastníkem,

Příklad:

```
cMessage* poz=new cMessage (“..“); // vlastníkem modul
cQueue fr;  fr . insert ( poz );  // poz ve vlastnictvi fr
cMessage* p1 = fr. pop ( );   // vyjmutý obj. ve vlastnictví modulu
```

- účel vlastnických vztahů:
  - pomůcka pro rušení objektů (garbage collector):
    - objekt může být zrušen výhradně jeho vlastníkem,
    - zrušením vlastníka se automaticky ruší i jím vlastněné objekty odvozené od třídy cObject,
    - ostatní objekty jsou systému neviditelné a musí být odstraněny explicitně v destruktoru příslušného modulu,

Vlastnictví objektů

– kontrola některých programátorských chyb:

- v modulu nelze odeslat, naplánovat nebo zrušit právě nevlastněný objekt (kontrola v send ( ), schedule ( ) i v destruktoru):
  - byl-li již odeslán jinému modulu,
  - byl-li odeslán „ sobě “, ale dosud nedorazil,
  - nebyl vyjmut z fronty nebo ze zprávy do které byl vnořen,

Příklad: vícenásobné odeslání zprávy

(broadcasting nebo retransmission)

```
// následuje odeslání zprávy msg do n směrů:
for ( int i = 0;  i < n - 1; i++ )
{
   cMessage * kopie = ( cMessage * ) msg -> dup ( );
   send ( kopie, “vystup“, i );   // na porty vystup [ i ]
}
   send ( msg, “vystup“, n-1 );   // na port vystup [ n-1 ]
```

- zjišťování vlastnických vztahů:

cObject* cObject :: owner ( ) const

….vrací odkaz na vlastníka objektu,

Příklad:

```
if ( this  == o.owner() ) { … je-li vlastnik }
                    else    {…..neni-li vlastnik }
```

Fronty

- cQueue: třída pro implementaci front (jako dvojitě zřetězených seznamů) objektů odvozených od třídy cObject (tj. cMessage, cPar,….)
- deklarace objektu fr třídy cQueue:

  cQueue fr (" fronta1");

Podpůrné funkce třídy cQueue:

int cQueue :: length ( ) ….vrací počet položek ve frontě,

bool cQueue :: empty ( )….. vrací true je-li fronta prázdná,

void cQueue :: insert (cObject * obj)

….. zařadí položku obj na konec fronty,

cObject* cQueue :: pop ( )……..vyjme první položku z fronty a vrací ukazatel na tuto položku

cObject* cQueue :: remove (cObject * obj)…vyjme položku obj a vrací ukazatel na ni

void cQueue :: clear ( )……vyprázdní frontu

void cQueue :: insertBefore (cObject* kam, cObject * obj)

…. zařadí položku obj do fronty před položku kam

void cQueue :: insertAfter (cObject* kam, cObject * obj)

……..zařadí položku obj do fronty za položku kam

cObject* :: head ( ) ….vrací odkaz na poslední položku fronty

cObject* :: tail ( ) ...vrací odkaz na první položku fronty

Příklad: systém hromadné obsluhy

```
// následuje obsah souboru     "celek. ned"
simple Generator          // typ generátor požadavků
   parameters:
      pocet_uloh,             // celkovy pocet pozadavků (viz   .ini)
      interval;               // intervaly mezi prichody (viz  .ini)
   gates:
      out: out;
endsimple

simple Likvidator            // typ likvidator uloh
   parameters:
      pocet_bunek  : numeric;  // počet buněk histogramu dob
   gates:
       in: in;
endsimple

simple Kanal                  // typ kanal s frontou FIFO
   parameters:
      doba_obsluhy : numeric,   // doba zpracovani ulohy (viz  .ini)
gates:
      in: in;
      out: out;
endsimple
```

Příklad: systém hromadné obsluhy

```
module SHO                          //  typ system hromadne obsluhy
   parameters:
      pocet_bunek : numeric,  // pocet bunek histogramu
   submodules:
      gen: Generator;         // instance typu Generator
            parameters:
            pocet_uloh = input ;    // interaktivni zadani nebo  .ini
      kan: Kanal;             // instance typu Kanal
      likvidator: Likvidator;
            parameters:
            pocet_bunek  = pocet_bunek ;   // pro histogram

   connections:       // vnitrni struktura SHO
      gen.out --> kan.in;
      kan.out --> likvidator.in;
endmodule

network sho : SHO              // sho: instance typu SHO
   parameters:
      // počet_bunek = input ( 15, "Pocet bunek pro histogram: ");
      // interaktivni vyzva a implicitni hodnota pro pocet bunek
endnetwork
// konec souboru “celek.ned“
```

Příklad: systém hromadné obsluhy

```
// následuje obsah souboru „celek.cpp“
# include <omnetpp.h>
class Likvidator : public cSimpleModule       // typ Likvidator
{   public: …;   // deklarace objektu pro statistiku ( viz později )
    virtual void initialize( );   // založení objektů pro statistiky
    virtual void handleMessage (cMessage * msg);  // likvidace
                  // pozadavku a shomaždování statistik
    virtual void finish( );       // výstup statistik  ( viz. později )
    Module_Class_Members ( Likvidator, cSimpleModule, 0 )
};

class Generator : public cSimpleModule
{ public:
   virtual void activity ( );   // procesove orientovany popis chovani
   Module_Class_Members ( Generator, cSimpleModule, 16384 )
};
class Kanal : public cSimpleModule
{  public:
    cMessage  *konec_obsluhy;    // ukazatel na vlastní zprávu
    cQueue fr;                   // deklarace fronty požadavků
    virtual void initialize( );
    virtual void handleMessage(cMessage * msg );
    Module_Class_Members ( Kanal, cSimpleModule, 0 )
};
```

Příklad: systém hromadné obsluhy

```
Define_Module ( Generator );                    // návaznost na NED
Define_Module( Kanal );
Define_Module( Likvidator );

void Generator :: activity()
{
   int  pocet_poz  = par ("pocet_uloh");   // parametry z NED
   cPar& interval  = par ("interval");
   for ( int i = 0; i < pocet_poz ; i++ )        // generovani požadavků
     {  char oznaceni_poz [32];
        sprintf ( oznaceni_poz, „pozadavek-%d", i );  // cislovani poz
        cMessage *poz = new cMessage ( oznaceni_poz );
        poz->setTimestamp( );   // nastaveni casove znamky požad.
        send( poz, "out" );          // odesláni požadavku pres port out
        wait( (double) interval ); // interval mezi prichody uloh
     }
}
void Kanal::initialize()
{
   //  následuje generování vlastní zprávy pro časování obsluhy
   konec_obsluhy = new cMessage ( „obsluha ukoncena„ );
   fr . setName ( “fronta“) ;
 }      // nastavení jména fronty
```

Příklad: systém hromadné obsluhy

```
void Kanal :: handleMessage (cMessage *msg)   // příjem zprávy
{  if ( ! msg == konec_obsluhy ) // příchod nového požadavku
    { ev<<„Nový příchod " << msg->name()<<endl;   // ladici text
      if ( fr . empty( ) )                    // volný kanal obsluhy
           {   fr . insert ( msg );
               simtime_t  dobaobsluhy =par ( “doba_obsluhy");
               scheduleAt
                        ( simTime() + dobaobsluhy, konec_obsluhy );
          }
          else   fr . insert ( msg );
    }
     else       //  příjem zprávy o ukončení obsluhy
             {
               ev <<"Obsluha ukoncena " << msg->name() << endl;
               cMessage * obslouzeny = ( cMessage * ) fr . pop();
               send ( obslouzeny,  “ out“ );   // odchod požadavku
               if ( ! fr . empty ( ) )
                     {
                       simtime_t dobaobsluhy = par ( “doba_obsluhy");
                       scheduleAt
                          ( simTime( ) + dobaobsluhy, konec_obsluhy );
                     }
             }
}
```

Příklad: systém hromadné obsluhy

```
void Kanal :: finish()
{ ……;  }     // výstup statistik o frontě

void Likvidator :: initialize()
{ ……;  }     // inicializace statistik

void Likvidator :: handleMessage ( cMessage *poz )
{ ……..;            // shromažďování statistik ( viz později )
  delete poz;      }

void Likvidator :: finish()
{ ……; }   // výstup skalárů, vektorů a histogramů
// konec soubor „ celek.cpp“
// následuje obsah souboru omnetpp.ini
[General]
network = sho
sim-time-limit = 100h
[Cmdenv]
[Tkenv]
[Run 1]
description=“toto je pouze nadpis výstupu na obrazovku"
sho.gen.pocet_uloh = 1000000
sho.gen.interval = exponential (1.25)
sho.kan.doba_obsluhy = exponential (1.0)
```

Příklad:modifikace SHO

poznámky: varianta příkladu s funkcí activity(), demonstrace použití jediného modulu.

Soubor .ned:

```
simple Kanal                    // typ kanal s frontou FIFO
   parameters:  doba_obsluhy, interval : numeric;
endsimple
network sho : Kanal     endnetwork
```

Soubor .cpp:

```
#include <omnetpp.h>

class Kanal : public cSimpleModule
{  public:
   virtual void activity();  // procesově orientovany popis
   Module_Class_Members ( Kanal, cSimpleModule, 16384 )   };
Define_Module( Kanal);
void Kanal :: activity()     // spusteni pri prijmu zpravy
{
   cQueue fr;                  // deklarace fronty
   cMessage *  prichod = new cMessage („prichod");
   cMessage *  odchod = new cMessage („odchod")
   scheduleAt( simTime(),  prichod);    // 1. příchod
```

Příklad:modifikace SHO

```
// pokračování funkce activity():
for ( ;; )
  {  cMessage* msg = receive();
     if ( (msg  == prichod)            // novy prichod
       {   simtime_t interval = par ("interval");
           scheduleAt ( simTime() + interval, prichod );
           cMessage* poz = new cMessage("poz");
           if ( fr.empty())              // prazdny kanal obsluhy
             {   fr. insert ( poz);
                 simtime_t doba_obsluhy = par ("doba_obsluhy");
                 scheduleAt ( simTime() + doba_obsluhy, odchod );
             }
           else  fr. insert ( poz);  //  obsluha pokracuje dalsiho pož.
       }
    if ((msg  == odchod )          // odchod pozadavku
       {    delete fr.pop();          // odchod pozadavku
            if ( ! fr. empty () )       // dalsi do obsluhy
                 { simtime_t doba_obsluhy = par ("doba_obsluhy");
                   scheduleAt ( simTime() + doba_obsluhy, odchod);
                 }
       }
   }       // konec cyklu for( ; ; )
}          // konec activity ()
```

Podpora automatů

- vhodné pro funkci handleMessage() při větším počtu přijímaných zpráv,
- existují 2 typy stavů:
  - stabilní: setrvání po určitou dobu simulačního času,
  - přechodný: pro zpřehlednění výpočtu,
- přijetí zprávy: přechod ze stabilního stavu přes případné přechodné stavy do dalšího stabilního stavu,
- oddělené specifikace funkce pro vstup a výstup stabilního stavu,

Způsob deklarace:

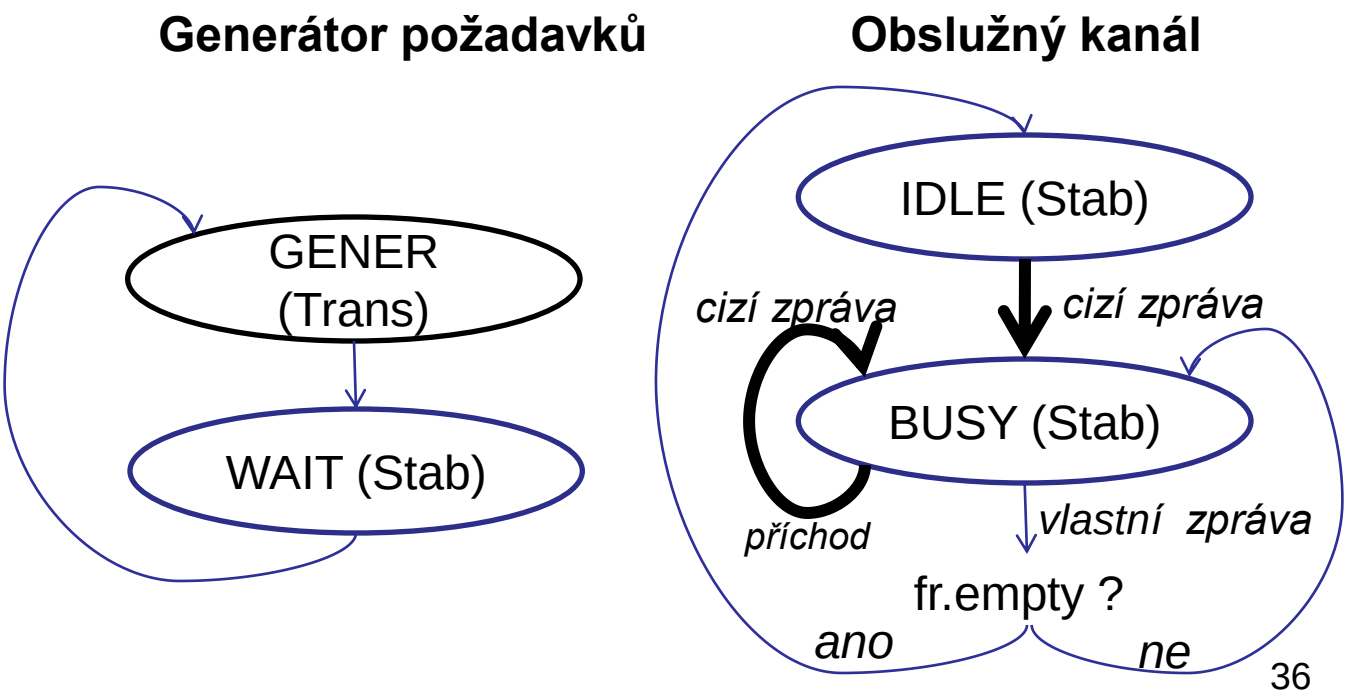
```
cFSM < identifikátor automatu >;  / /     deklarace objektu
// následuje výčet všech stavů
enum   {
            //  deklarace  stabilních stavů
           < identifikátor stab. stavu> = FSM_Steady (1);
           < identifikátor stab. stavu> = FSM_Steady (2);
            ……;
            //  deklarace  přechodných stavů
            < identifikátor prechod. stavu> = FSM_Transient (1);
           < identifikátor prechod. stavu> = FSM_Transient (2);
            ……;
        }
```

Podpora automatů

Forma popisu přechodové a výstupní funkce:

```
void <identifikátor třídy> :: handleMessage (cMessage *  msg)
{   FSM_Switch ( <identifikátor automatu> )
        {  case FSM_Enter ( <identifikátor  stavu> ) :
                ……;   //  operace  v čase vstupu
                break;
           case FSM_Exit ( <identifikátor  stavu> ) :
                ……;  //  operace v čase výstupu
                FSM_Goto ( <ident. automatu>, <ident. stavu>)  break;
            ……..; // specifikace zbývajících stavů
        }   // konec  popisu automatu
 }       // konec   hanleMessage
```

Příklad: automatová specifikace jednoduchého SHO



Příklad: automatový popis SHO

```
 class Kanal : public cSimpleModule
{
   public:
   cMessage *konec_obsluhy;      // indikace konce obsluhy
   cQueue fr;                    // deklarace fronty

   cFSM kan;                        // automat pro kanal
   enum  {
             IDLE = FSM_Steady(0),
             BUSY = FSM_Steady(1),
           };
   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
   virtual void finish() ;
   Module_Class_Members ( Kanal, cSimpleModule, 0 )
}

void Kanal::initialize()
{
   // nasleduje vlastni zprava pro indikaci konce obsluhy
   konec_obsluhy = new cMessage("obsluha ukoncena");
   FSM_Goto ( kan, IDLE);  // nastavení počátečního stavu
}
```

Příklad: automatový popis SHO

```
void Kanal::handleMessage(cMessage *msg)
{   FSM_Switch (kan)
 {  case FSM_Enter (IDLE):    break;  // žádná operace
    case FSM_Exit (IDLE):
                    fr. insert ( msg);  // příchod požadavku
                     FSM_Goto (kan, BUSY);   break;

    case FSM_Enter (BUSY):
        if (! konec_obsluhy -> isScheduled() )
        {  simtime_t   doba_obsluhy = par ("doba_obsluhy");
           scheduleAt  ( simTime() + doba_obsluhy, konec_obsluhy );
         }   break;

    case FSM_Exit (BUSY):
      if ( ! msg -> isSelfMessage())              // příchod požadavku
                    { fr. insert(msg);  break; }  // pozadavek do fronty
             else  // prijem vlastni zpravy (o ukonceni obsluhy
                    {  cMessage* obslouzeny = ( cMessage* ) fr. pop ();
                       send ( obslouzeny, "out" );  // odchod požadavku
                        if ( ! fr. empty () )   break;
                                    else  {  FSM_Goto (kan, IDLE); break; }
                    }
  }
}
```

Příklad: automatový popis SHO

```
class Generator : public cSimpleModule
{
   public:
   cFSM gen;      // automat pro generator
   enum {
              GENER = FSM_Transient(1),
               WAIT = FSM_Steady(1),
            };
   cMessage* endofWait;          // indikace konce intervalu
   double interval;
   virtual void initialize();
   virtual void handleMessage(cMessage *poz);
   Module_Class_Members ( Generator, cSimpleModule, 0 );
};

void Generator::initialize()
{
  interval = par("interval");
  cMessage* start = new cMessage ("start");
  FSM_Goto (gen,GENER);
  endofWait = new cMessage ("konec_intervalu");
  scheduleAt (simTime(), start);       // aktivace handleMessage()
}
```

Příklad: automatový popis SHO

```
void Generator::handleMessage(cMessage *poz)
{
   FSM_Switch (gen)
   {
    case FSM_Exit (GENER):
                // v tomto stavu automat setrvá po nulovou dobu ( z
                //  hlediska simulačního času)
                // operace: generování požadavku
                cMessage *poz = new cMessage( …);
                send( poz, "out" );         // odeslani  pozadavku
                FSM_Goto (gen,WAIT);
                break;

    case FSM_Enter (WAIT):
                // operace: určení času výstupu ze stavu WAIT
                scheduleAt( simTime() + interval, endofWait );
                break;

    case FSM_Exit (WAIT):
                // prichazi pouze vlastni zpráva  (interval vyčerpán)
                FSM_Goto (gen, GENER);
                break;
   }
}
```

Porty: identifikace a zjišťování parametrů

- porty modulu: objekty třídy cGate,

  cGate* cModule :: gate (const char* jmeno)

  vrací ukazatel na port identifikovaný jménem,

  cGate* cModule :: gate (const char* jmeno, int index)

  vrací ukazatel na port identifikovaný jménem vektoru a indexem,

  int cModule :: gateSize (const char* jmeno)

  vrací počet portů vektoru, v případě nespecifikovaného vektoru vrací 0; nejde-li o vektor vrací 1

  int cModule :: findGate (const char* jmeno)

  vrací jednoznačné identifikační číslo portu specifikovaného jménem,

  int cModule :: findGate (const char* jmeno, int index)

  vrací jednoznačné identifikační číslo portu specifikovaného jménem vektoru a indexem,

  int cModule :: index( )

  pokud je instance modulu součástí vektoru, pak vrací její pozici, jinak vrací 0

Porty: identifikace a zjišťování parametrů

int cGate :: index ( ) …. vrací index konkrétního portu ve vektoru; nejde-li o vektor vrací 1,

int cGate :: id ( ) …. …..vrací jednoznačnou identifikace portu v poli všech portů modulu,

int cGate :: size ( ) …. . vrací dimenzi vektoru portů, v případě nespecifikovaného vektoru vrací 0; nejde-li o vektor vrací 1,

char cGate :: type ( ) …. vrací znak " I " jde-li o vstupní bránu, jinak vrací znak "O " ,

Příklady použití:

- předpoklad: **simple** M1
  parameters: , ……..; **gates**: **out**: vyst [3];
  endmodule

```
cGate* pport = gate ( " vyst" , 1 );   // pointr na vyst [ 1 ]
       send ( msg, pport );
int    ident1 = pport ->  index ( );          // vrací 1
       send ( msg, vyst, ident1 );        // jako předchozí send
cGate* pport1 = gate ( " vyst" );          // vrací pointr na vektor
int    dimenze = pport1->  size ( );       // vrací 3
int    ident2 = findGate ( "vyst", 0 );   //vrací identifikaci vyst [ 0 ]
       send ( msg, ident2) ;   // jako send ( msg, vyst, 0)
```

Vlastnosti komunikačních linek

- uloženy v instanci podtřídy cBasicChannel odvozené od třídy Channel,
- instance třídy cChannel je dostupná pomocí metody channel () výstupního portu:

  cChannel *cGate :: channel ( )…vrací ukazatel na objekt základní třídy cChannel,
- přetypování:

  cBasicChannel * check_and_cast < cBasicChannel* > (Channel *)

  …….vrací ukazatel na objekt podtřídy cBasicChannel v němž jsou uloženy atributy komunikační linky

Příklad: předpoklad: **simple M1**
gates: **out**: vystup ;
endmodule

- čtení parametrů linky připojené na výstupní port vystup:

```
cGate* pvyst = gate ( "vystup" );  // pointer na port vystup
cBasicChannel * k1 = check_and_cast <cBasicChannel*>
                                    ( pvyst -> channel ( ) );
// následuje použití metod třídy cBasicChannel:
double d = k1 -> delay ( ) ;      // zpoždění linky
double e = k1 -> error ( ) ;      // pravděpodobnost chyby
double r = k1 ->  datarate ( ) ; // přenosová kapacita
```

Zjišťování konektivity portů

bool cGate :: isConnected ( ) ….vrací true pokud je daný port připojen, jinak false

cGate * cGate :: toGate ( ) ……..vrací ukazatel vstupního portu, který je spojen s daným výstupním portem; jde-li o vstupní port jednoduchého modulu, pak vrací NULL

cGate * cGate :: fromGate ( ) ….. vrací ukazatel výstupního portu, který je spojen s daným vstupním portem; jde-li o výstupní port jednoduchého modulu, pak vrací NULL

Příklad:

```
cGate * pg1 = gate (“g1“) ;   // nastavení ukazatele na port g1
if ( pg1 -> isConnected ( ) )
   {  cGate * soused = ( pg1 -> type ( ) == ´O´) // port g1 připojen
                       ? pg1 -> toGate ()   // následující vstup
                       : pg1-> fromGate (); // předcházející výstup
   }
     else   {…….}          // port g1 není připojen
```

Funkce pro přístup k modulu (vlastníku portu):

cModule * cGate :: ownerModule ( )…..vrací ukazatel na modul v němž je daný port specifikován

Zjišťování konektivity portů

Příklad: zjišťování cílového jednoduchého modulu:

```
cGate * p1 = gate ( " vystup"  );        // nastavení ukazatele
while ( p1 -> toGate ( )   !=  NULL )   p1 = p1 -> toGate ( ) ;
cModule * cil = p1 -> ownerModule ( ) ;    // cílový modul
```

Funkce pro zjišťování stavu portů:

bool cGate :: isBusy ( ) ….vrací true pokud je přes daný port vysílaná zpráva

simtime_t cGate :: transmissionFinishes ( ) ….vrací absolutní čas ukončení přenosu přes port pokud je busy

Příklad: čekání na volnou komunikační linku

```
cMessage * pkt = new cMessage (….);
pkt -> setLength ( …… );                // nastavení délky paketu
cGate * p1 = gate ( " vystup"  );        // nastavení ukazatele

if ( p1 -> isBusy ( ) )     // test na obsazenost portu vystup
     // následuje čekání na uvolnění linky
    wait ( p1-> transmissionFinishes ( ) – simTime( ) ) ;

send ( pkt,  " vystup" );          // zahájení vysílání paketu
```

Příklad: server pro vysílání rámců

Způsob řešení: zjišťuje se čas ukončení vysílání rámce pomocí funkce transmissionFinishes(), konec vysílání je indikován přijetím vlastní zprávy (tj.instance endSend)

```
void Server::handleMessage(cMessage *msg)
{  if ( ! msg -> isSelfMessage() )    // příjem rámce na vyslání
  {  if ( fr.empty())                  // právě se nevysílá
        { fr.insert (msg);
          send ((cMessage*)  fr.tail() -> dup();, "out");  // odeslání
          simtime_t  ukonceni =    // čas ukončení vysíláni
            ( gate("out") -> transmissionFinishes() == 0) ?simTime() ;
                                  : gate("out") -> transmissionFinishes( );
           scheduleAt( ukonceni, &endSend );  // indikace ukončení
        }  else  fr. insert (msg);    // přijatý rámec bude čekat
   }
 else { delete fr.pop();             // kopie byla odeslana
         if ( ! fr.empty())
            { send ((cMessage*) fr.tail() -> dup();, "out");
              simtime_t  ukonceni =    // čas ukončení vysíláni
             (gate("out")->transmissionFinishes() == 0) ?simTime() ;
                                  : gate("out") -> transmissionFinishes( );
              scheduleAt ( ukonceni, &endSend );
            }
       }
}
```

Identifikace modulů

- int cModule :: id ( ) …….vrací identifikační číslo modulu,
  př.: int idmod = id ( ) ; // identifikační číslo daného modulu

- metoda module globálního objektu simulation vrací ukazatel na modul určený identifikačním číslem,
- cModule* pmod = simulation . module (idmod);

- int cModule::findSubmodule (const char* jméno, int index =-1) ……vrací identifikaci submodulu specifikovaného (v daném složeném modulu) jménem a případně indexem, nebyl-li nalezen vrací -1

  int id_submod_m1 = pmod -> findSubmodule (“m1“);

- cModule* cModule :: submodule (const char* jméno, int index =-1)
  vrací ukazatel na submodul specifikovaný (v daném modulu) jménem a případně indexem, nebyl-li nalezen vrací -1

- cModule* cModule :: parentModule ( )……vrací pointer na nadřazený modul
- Příklad: přístup k parametrům rodičovského modulu:

double doba_zpracovani = parentModule ( ) -> par (“ doba“);

Filosofie zpráv

- objekty třídy cMessage,
- použití: reprezentace událostí, zpráv, paketů, požadavků, atd.,

Standardní atributy objektů třídy cMessage:

```
const char * name;   // jméno zprávy ( pro ladění )
int kind;            // hodnoty ≥ 0 volně k použití,
int length;          // délka zprávy ( ovlivňuje dobu vysílání )
int priority;        // priorita ( ovlivňuje plánování do FEL )
boolean bit_error_flag   // příznak chyby - je automaticky
      generován při vysílání zprávy s pravděpodobností
       p = 1-(1-ber)^length
simtime_t  timestamp   //  čas. známka (k dispozici uživatele)
```

Nastavení hodnot standardních atributů:

- pomocí parametrů konstruktoru při generování instancí třídy

```
cMessage :: cMessage ( const char* name, int k, long ln, int pri,
 bool err ) : cObject ( name )
```

- pomocí funkcí třídy cMessage:
  - void cMessage :: setKind (int k) ;
  - void cMessage :: setLenght (long l),
  - void cMessage :: setPriority (int k),
  - void cMessage :: setBitError (bool err),
  - void cMessage :: setTimeStamp (simtime_t t)…pokud je vynechán akt. parametr, pak hodnota časové známky = okamžitá hodnota simulačního času,

Filosofie zpráv

- čtení hodnot standardních atributů: pomocí funkcí:

```
int         cMessage :: kind ( );
long        cMessage :: lenght ( );
int         cMessage :: priority (..);
bool        cMessage :: hasBitError ( );
simtime_t   cMessage :: timestamp ( );
```

simtime_t cMessage :: creationTime ( ) // čas vytvoření zprávy
simtime_t cMessage :: sendingTime ( ) // čas odeslání zprávy ,
simtime_t cMessage :: arrivalTime ( ) // čas přijetí zprávy

- _každá zpráva vlastní universální ukazatel (void *), který lze použít pro různé účely (nejčastěji jako pointer na strukturu, která nějak souvisí s vlastní zprávou a tak umožňuje její snadnější identifikaci v případě většího počtu vlastních zpráv)

void cMessage :: set ContextPointer (void *p)
…….nastaví pointer na libovolnou hodnotu
void* cMessage :: contextPointer (void *p)
…….vrací hodnotu contex pointru dané zprávy,

- testování zpráv:

bool cMessage :: isSelfMessage ( ) ..vrací true je-li zpráva vlastní
bool cMessage :: isScheduled ( )
……..vrací true jde-li o dosud náplánovanou vlastní zprávu,

Příklad:modifikace SHO

poznámky: jiná varianta příkladu : demonstrace použití setContextPointer() a contextpointer(); požadavek v obsluze není ve frontě, ale bude připojen k vlastní zprávě „konec_obsluhy“

```
void Kanal :: handleMessage (cMessage *msg)   // příjem zprávy
{  if ( ! msg -> isSelfMessage() )       // příchod nového požadavku
    {  if ( ! konec_obsluhy -> isScheduled() ) // volný kanal obsluhy
          {   konec_obsluhy ->setContextPointer ( msg ) ;
              simtime_t  dobaobsluhy =par ( “doba_obsluhy");
              scheduleAt
                    ( simTime() + dobaobsluhy, konec_obsluhy ); }
         else   fr . insert ( msg );
    }
     else       //  příjem zprávy o ukončení obsluhy
        { cMessage * obslouzeny = (cMessage*) konec_obsluhy ->
                                                  contextPointer();
          send ( obslouzeny,  “ out“ );   // odchod požadavku
               if ( ! fr . empty ( ) )
                   { konec_obsluhy ->
                           setContextPointer((cMessage*) fr.pop());
                     simtime_t dobaobsluhy = par ( “doba_obsluhy");
                     scheduleAt
                       (simTime( ) + dobaobsluhy, konec_obsluhy ); }
        }
}
```

Příklad: server pro vysílání rámců

Způsob řešení:

- vypočte se doba vysílání z délky rámce a přenosové kapacity linky ,
- využívá se funkce waitAndEnqueue (..)

```
void Server:: activity()
{     cMessage* msg;
  W: msg = receive();          // není co vysílat
  S:  send (msg, "out");        // odeslání paketu po přijetí
        // pointer k na kanal umožní přístup k přenosové rychlosti
         cBasicChannel * k = check_and_cast<cBasicChannel*>
                                              ( gate ("out") -> channel() ) ;
        // nasleduje vypocet doby vysilani: del
        double del =  msg -> length()  /  k -> datarate();
        waitAndEnqueue ( del,  &fr);  // pozastaveno behem vysilani
        // funguje i bez wait, ale casy odeslani jsou dany okamzikem
        // provedeni funkce send()
        if (! fr.empty())
                    {  // pokračuje vysílání nashromážděných
                       // paketů
                        msg = (cMessage*) fr. pop();
                         goto S;
                     }
              else goto W;
}
```

Filosofie zpráv: pokračování

cModule* cMessage :: senderModule () …. vrací pointer na modul, který zprávu vyslal (NULL, pokud nebyla odeslána),

int cMessage :: senderModuleId () …. vrací identifikační číslo modulu, který zprávu vyslal (-1 pokud nebyla odeslána),

cGate* cMessage :: senderGate () …. vrací pointer na port přes který byla zpráva poslána (NULL, pokud nebyla odeslána nebo jde o vlastní zprávu),

int cMessage :: senderGateId () …. vrací indentifikační číslo portu přes který byla zpráva odeslána (-1 pokud nebyla odeslána),

cGate* cMessage :: arrivalGate () …. vrací pointer na port přes který byla zpráva přijata (NULL, pokud nebyla odeslána),

int cMessage :: arrivalGateId () …. vrací identifikační číslo portu přes který byla zpráva přijata (-1 nebyla-li odeslána nebo vlastní zpráva),

bool cMessage :: arrivedOn (int id) …. vrací true pokud byla zpráva přijatá přes port s identifikací id ,

bool cMessage :: arrivedOn (const char* jmeno, int index = 0) …. vrací true pokud byla zpráva přijatá přes port specifikovaný jménem a indexem,

Filosofie zpráv: pokračování

Vnořování zpráv do jiných zpráv:

- void cMessage :: encapsulate (cMessage * data),
  - do dané zprávy vloží zprávu odkazovanou pointrem data,
  - modifikuje délku dané zprávy o délku vkládané zprávy,

Poznámka: zpráva může obsahovat jednu vloženou zprávu,

Příklad:

```
cMessage * data = new cMessage ( "…") ;
cMessage * tcpseg = new cMessage ( "…" ) ;
 tcpseg -> encapsulate ( data );     // vloží data do tcpseg
```

Rozbalování zpráv:

- cMessage* cMessage :: encapsulatedMsg ( ),
  - vrací ukazatel na zprávu vloženou do dané zprávy; není-li taková pak vrací NULL

- cMessage* cMessage :: decapsulate ( ),
  - z dané zprávy vyjme vloženou zprávu a vrací odkaz na tuto zprávu,
  - vrací NULL pokud žádná zpráva nebyla vnořena,

  příklad: cMessage * data = tcpseg -> decapsulate ( );
  // z tcpseg vyjme data

Filosofie zpráv: pokračování

Připojování nových datových struktur

- dvě možnosti:
  - starší způsob: připojením objektů třídy cPar,
  - novější způsob: definicí zprávy v souboru “ jméno . msg“ spolu s využitím speciální podpory „message compiler“ , která automaticky generuje příslušné podtřídy třídy cMessage spolu s funkcemi pro nastavení i čtení každé připojené položky,
- je možné vytvářet více zpráv s různou strukturou.

Příklad: obsah souboru „ jmeno . msg “:

```
message ramec     // vnitřní struktura rámce
{   // následuje specifikace položek ramce
    fields:  int source;
             int destination;
             ……..   } ;
message paket       // vnitřní struktura paketu
{  fields:    int cilova_adr;   // specifikace položek dat
              string data;     } ;
```

- vnoření instancí třídy paket do instancí třídy ramec (v programu příslušného modulu) :

```
paket*   pkt = new paket (  “p1“ );  // vytvoření paketu
ramec*     r = new ramec ( “r1“ );      // vytvoření rámce
           r -> encapsulate ( pkt );   // zabalení paketu
```

Filosofie zpráv: pokračování

- přípustné typy dílčích položek zprávy:
  - primitivní typy : **bool, char, short, unsigned short, int, unsigned int, long, unsigned long, double**
  - omezená, resp. neomezená pole primitivních typů:
    příklad: **double** pocty [4]; **double** soucty [ ] ;
  - řetězce, př. **string** jmeno;
  - instance tříd a struktur,

- byla-li zpráva definována v souboru „jmeno . msg“, pak message compiler vygeneruje tyto soubory:
  - soubor „jmeno_m . h“ (deklarace hlavičky podtřídy odvozené od třídy cMessage),
  - soubor „jmeno_m . cpp“ s funkcemi pro nastavování a získávání hodnot všech specifikovaných položek zprávy,
  - v případě instancí třídy paket (viz minulý slide):
    virtual int getCilova_adr ( ),
    virtual void setCilova_adr (int cil),
    virtual int getData( ),
    virtual void setData (string str),

Příklad použití:

```
paket * pkt = new paket ( “data1”);
        pkt -> setCilova_adr ( 1024 ) ;   // nastavení adresy
        pkt -> setData ( “0AED” );        // nastavení dat
```

Filosofie zpráv: pokračování

- v případě omezeného pole double pocty [4] by šlo o funkce:

```
virtual double getPocty ( unsigned k );          // čtení k-té položky
virtual void setPocty ( unsigned k, double d );  // zápis položky
virtual unsigned getPoctyArraySize (  );         // čtení délky pole
```

- v případě neomezeného pole int poct y [ ] by šlo o funkce:

```
virtual double getPocty ( unsigned k ); // čtení k-té položky
virtual void setPocty ( unsigned k, int d );    // zápis položky
virtual void setPoctyArraySize (unsigned n); // nastavení
                                              // dimenze
virtual unsigned getPoctyArraySize (  );      // čtení dimenze
```

Kopírování zpráv:

virtual cPolymorphic* cObject :: dup ( )

….vytvoří duplikát příslušné zprávy a vrátí ukazatel na tento duplikát

příklad: kopie zprávy odkazované ukazatelem msg

```
cMessage * kopie1 = ( cMessage* ) msg -> dup ( );
```

- jiná možnost: použití konstruktoru

```
cMessage * kopie2 = new cMessage ( *msg );
```

Model přepínače

Charakteristika:

- přepínač propojuje jednotlivé počítače v lokální síti Ethernet a umožňuje plně duplexní provoz,
- směrování přicházejících rámců je zajišťováno jediným procesorem ; pokud nemůže být rámec ihned tímto procesorem zpracován tak je zařazen do vstupní fronty,
- z přícházejících rámců přepínač dynamicky vytváří směrovací tabulku připojených počítačů a jim odpovídajících portů,
  - předpoklad: rámce obsahují kromě standardních atributů také zdrojovou a cílovou adresu:

```
message PAKET
{  fields: int zdr_adr;
           int cil_adr;
};
```

  - rámce určené počítači, jejichž adresa se nachází ve směrovací tabulce jsou směrovány pouze přes příslušný port, ostatní rámce jsou směrovány do všech zbývajících směrů (s výjimkou směru příchozího),
  - směrovací tabulka je periodicky redukována o záznamy počítačů, které od poslední redukce nevyslaly žádný rámec,
- každý výstupní port přepinače má svou výstupní frontu a je buzen samostatným procesorem (server); doba vysílání je dána délkou rámce a přenosovou rychlostí dané linky,

Vnitřní struktura přepínače

Dílčí moduly přepínače:

```
simple SMEROVAC
   parameters: doba_smerovani, odmlceni_pocitace: numeric;
   gates: in: vst [ ];  out: vyst [ ];
endsimple

simple SERVER
   parameters: delka_IFG : numeric;
   gates: in: od_smer;   out: physicalOut;
endsimple
```

Vnitřní struktura přepínače:

```
module SWITCH
    gates:  in: vst [ ];    out: vyst [ ] ;
submodules:  sm:      SMEROVAC ;
                    gatesizes: vst [sizeof (vst)], vyst [sizeof (vyst)] ;
                    server: SERVER [ sizeof (vyst) ] ;
connections:
          for i = 0 .. sizeof (vst)  -1 do
          vst [ i ]          -->  sm.vst [ i ];
          sm.vyst [ i ]     -->  server [ i ] . od_smer;
          server [ i ] . physicalOut --> vyst [ i ];
          endfor;
endmodule
```

Model směrovací části přepínače

```
class SMEROVAC : public cSimpleModule
{  public:
   cQueue fr;
   class Sireni   {  public:  int       portindex;   // port příchodu
                              simtime_t cas;      // čas příchodu
                   };
   typedef std:: map < int, Sireni*> SmerTab; // typ pro směr. Tab.
   SmerTab tab;        // směrovací tabulka
   int pocet_portu;    // celkový počet portů směrovače
   cMessage* er; // pro časování periodických redukcí směr. tab.
   double odmlceni_pocitace;    // parametr z NED
   double doba_smerovani;       // parametr z NED
   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
   Module_Class_Members (PREPINAC, cSimpleModule, 0 )
};

void SMEROVAC :: initialize()
{  odmlceni_pocitace = (double) par ("odmlceni_pocitace");
   doba_smerovani       = (double) par ("doba_smerovani");
   er = new cMessage ("kontrola starych zaznamu");
   scheduleAt ( simTime() + odmlceni_pocitace,    er);
   pocet_portu = gateSize ("vst");
}
```

Model směrovací části přepínače

```
void SMEROVAC::handleMessage( cMessage *msg )
{  PAKET* pkt;                          // ptr na příchozí paket
   int adr;                             // adresa zdroje
   if ( ! msg ->isSelfMessage())        // prijem noveho paketu
     { pkt = check_and_cast <PAKET*> (msg);
       adr = pkt -> getZdr_adr();       // adresa zdrojoveho pocitace
       SmerTab :: iterator it;
       it = tab.find(adr);        // vyhledání  adresy
       if ( it == tab.end() )   // zdrojova adresa dosud neni v tabulce
              {  // vlozeni zdroje a prislusneho portu do tabulky
                Sireni* s = new Sireni;
                 cGate* g  = pkt -> arrivalGate();   // ptr na prich. port
                 s -> portindex = g -> index();  // index vstupniho portu
                 // cas vlozeni zaznamu
                  s -> cas = simTime();
                 tab [adr] = s;     }                  // udaj pro zdroj
      else  {  // zdrojova adresa je v tabulce,  opravi se pouze cas
                 Sireni* s = (*it) . second;
                 s -> cas = simTime();       }
      if (fr.empty()) scheduleAt( simTime() +doba_smerovani, pkt)
                    else  fr.insert (pkt);  // smerovaci procesor busy
     }  // konec činností souvisejících s příjmem nového paketu
```

Model směrovací části přepínače

```
else if ( msg == er)
        {  // vlastni zprava:  časuje výmaz starých zaznamů
           SmerTab :: iterator it ;
           for ( it = tab . begin();  it != tab.end();  it++ )
              {
                Sireni* s = (*it).second;
                // nasleduje výmaz staršího záznamu
                 simtime_t delta = simTime()  - s ->cas;
                 if ( delta >= odmlceni_pocitace )  tab . erase (it);
              }
           // naplanuj pristi procisteni  smerovaci tabulky
          scheduleAt (simTime() + odmlceni_pocitace, er);
        }  // konec aktualizace směrovací tabulky
       else  {
                // prijem ramce jako vlastni zpravy:
                 // dokonceni smerovani,  zacne vlastni vysilani
                pkt = check_and_cast <PAKET*> (msg);
                 int adr_cile = pkt->getCil_adr();  // čtení cil. adresy
                 ev << "urceno pro stanici " << adr_cile << endl;
                // nasleduje vyhledani prislusneho vystupniho portu
                 SmerTab :: iterator it;
                 it = tab . find (adr_cile);          // vyhledání adresy
```

Model směrovací části přepínače

```
if (it == tab . end())   // cilova adresa  neni v tabulce
      { // bude broadcast  ( kromě příchozího portu )
        int zdr = pkt -> getZdr_adr();
        it = tab.find (zdr);
        Sireni* s = (*it) . second;
        int vst_port = s -> portindex; // příchozí port
        for ( int i = 0; i < pocet_portu ; i++)
              { cMessage* kopie =(cMessage*) pkt->dup();
                if ( ! (i == vst_port) )  send ( kopie, "vyst", i);
              }
        delete pkt;        // zrušení originálu
        ev << "odeslano do vsech smeru " << endl;
      }  //  konec  broadcast
      else  // následuje  cilove odeslani na port, který
          směřuje ke stanici s adresou adr_cile
          {  ev << "odeslano adresne" << endl;
             Sireni* s = (*it) . second;
             int p = s -> portindex;
              send ( pkt, "vyst", p );   // adresni smerovani
          }   //  konec  směrování
if ( ! fr.empty())   // fronta ramcu cekajicich na smerovani
      {  PAKET *p = (PAKET*) fr.pop();
         scheduleAt( simTime() + doba_smerovani, pkt);
      }  // začátek dalšího směrování
   }   // konec aktivit souvisejících s příjmem vl. zprávy
}   //  end of handleMessage
```

Model vysílací části přepínače

```
 class SERVER: public cSimpleModule
{  private: cQueue fr;   // fronta paketu  čekajících na odeslání
            double delka_IFG; // delka mezery mezi rámci
            cFSM wr;     // automat pro vysilani
            enum {
                   IDLE_W = FSM_Steady(0),
                   IFG = FSM_Steady(1),
                  TRANSMITTING = FSM_Steady(2),
                 };
   // nasleduji pointry na vlastni zpravy zajistujici vnitrni casovani
   cMessage * endTxMsg;   //oznámí konec vysílání rámce
   cMessage * endIFGMsg; //oznámí konec mezirámcové mezery
   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
   Module_Class_Members ( SERVER, cSimpleModule, 0 );
};

 void SERVER : :initialize()
{
  endTxMsg      = new cMessage ("konec vysilani");
  endIFGMsg     = new cMessage ("konec mezery");
  delka_IFG = (double) par("delka_IFG");     // parametr z NE
  FSM_Goto (wr,IDLE_W);       // počáteční stav
}
```

Model vysílací části přepinače

```
 void SERVER::handleMessage (cMessage *msg)
{
  FSM_Switch (wr)          // specifikace automatu pro zapis
    {
      case FSM_Enter (IDLE_W): break // žádná operace

      case FSM_Exit (IDLE_W):
      // příjat rámec  od směrovače k odeslání,  jiná možnost není
            fr . insert (msg);
            FSM_Goto (wr, IFG);
            break;

      case FSM_Enter (IFG):
            // následuje kontrola ukončení mezirámcové mezery
            if ( ! endIFGMsg -> isScheduled())
                  scheduleAt( simTime() + delka_IFG, endIFGMsg);
           break;

            // během čekání na konec mezirámcové mezery může
             // být přijat  nový rámec od směrovače a v důsledku
            // toho  by nastal výstup FSM_Exit (IFG) následovaný
           // vstupem   FSM_Enter (IFG); opětovné plánování již
           //  naplánované zprávy by vyvolalo chybu
```

Model vysílací části přepinače

```
    case FSM_Exit (IFG):
        if (msg->arrivalGate() == gate("od_smer"))  // nový rámec
              {  fr.insert( msg);  break; }
        if (msg == endIFGMsg)     // konec mezery
          {  // nasleduje zacatek fyzickeho vysilani ramce
             send (( cMessage *) fr.tail()-> dup(), "physicalOut");
             simtime_t ukonceni =
               (gate("physicalOut") -> transmissionFinishes() == 0)
                    ? simTime()
                    :  gate("physicalOut") -> transmissionFinishes() ;
             scheduleAt ( ukonceni, endTxMsg );  // konec vysilaní
             FSM_Goto(wr,TRANSMITTING);
             break;
          }
     case FSM_Enter (TRANSMITTING):  break;
     case FSM_Exit (TRANSMITTING):
      if (msg == endTxMsg)       // konec vysílání
                    {   if (fr.empty ())   FSM_Goto(wr, IDLE_W);
                                    else   FSM_Goto(wr, IFG);
                        break;   }
      if (msg->arrivalGate() == gate("od_prep"))  break;
        // příjem nového rámce během vysílání
   }      // konec automatu wr
}         // konec handleMessage
```

Příklad: model sběrnice

předpoklad: stanice jsou ekvidistantně rozloženy podél sběrnice

simple Bus **parameters:** doba_sireni; **//** od portu k portu
gates: in: vst [ ]; **out:** vyst [ ]; **endmodule**

```
// následuje funkční popis:
class Bus : public cSimpleModule
{  public:  class Sireni
          { public:  int port;    // příští port vystupu při šíření signálu
                     char* smer;            // smer sireni po sbernici
                     Sireni (int, char*);   // konstruktor
           };
          int          pocet_portu;    // celkový počet portů sběrnice
          double     doba_sireni;    //  od portu k portu
          virtual void initialize();
          virtual void handleMessage(cMessage *msg);
          Module_Class_Members (Bus, cSimpleModule, 0 )     };

Bus:: Sireni :: Sireni ( int p, char* sm )
                    {   port = p;   smer = sm;     };

void Bus :: initialize()
{   pocet_portu  =  gateSize ("vst");
    doba_sirení  =  (double) par ( “doba_sireni”);
    if (! pocet_portu == gate("vyst")  -> size() )   // kontrola
                          ev << "chyba v portech" << endl;     }
```

Příklad: model sběrnice

```
void Bus::handleMessage  ( cMessage *msg )
{   if (! msg ->isSelfMessage())       // příjem nové zpravy
   {  int vstup_port = msg ->arrivalGate() -> index();
      if (vstup_port == 0)   // nasleduje sireni zpravy doprava
             {  Sireni* ptr = new Sireni ( vstup_port+1, "Right");
                msg -> setContextPointer ( ptr );  // smer sireni
                scheduleAt ( simTime() + doba_sireni, msg);
             }
      else  if (vstup_port == gateSize("vst") -1 ) // sireni doleva
                    {  Sireni* ptr = new Sireni ( vstup_port-1, "Left");
                       msg -> setContextPointer ( ptr );  // smer sireni
                       scheduleAt ( simTime() + doba_sireni , msg);
                    }
              else {  // nasleduje sireni zpravy doleva i doprava
                       cMessage* right = (cMessage*) msg -> dup();
                       Sireni* ptr1 = new Sireni ( vstup_port+1, "Right");
                       right -> setContextPointer (ptr1);  // smer sireni
                       Sireni* ptr2 = new Sireni ( vstup_port-1, "Left");
                       msg -> setContextPointer (ptr2);  // smer sireni
                       scheduleAt ( simTime() + doba_sireni, right);
                       scheduleAt ( simTime() + doba_sireni, msg);
                    }
   }   // konec zpracování nové zprávy
```

Příklad: model sběrnice

```
  else  { // sirici se zprava dorazila k dalsimu portu
            cMessage* d = (cMessage*) msg -> dup();
            // následuje určení příslušného výstupního portu
            Bus :: Sireni* s = (Sireni*)msg -> contextPointer();
            // následuje odeslaní duplikatu d na port
            send ( d, "vyst",  s -> port );
            ev << "duplikat vyslan na port  " << s -> port << endl;
            if ( s -> port == 0 || s -> port == pocet_portu -1 )
                        {
                          delete msg;
                           ev << "sireni dorazilo na konec" << endl;
                           return;
                        }
                   else {
                           if ( s -> smer == "Left")  s -> port  -- ;
                                                 else  s -> port ++ ;
                           // sireni zpravy pokracuje
                           scheduleAt ( simTime() + doba_sireni,  msg);
                        }
             }
}     // konec  handleMessage90
```

Příklad: CSMA / CD

Zjednodušený nástin koncepce řešení v systému INET:

Příjem počátku rámce ze sítě:

stav **RECEIVING**,

naplánování konce příjmu :

dle délky rámce a přenos. kapacity linky

(vl. zpráva endRxMsg, kind = ENDRECEPTION)

Začátek odesláni rámce do sítě:

stav **TRANSMITTING**,

naplánování konce vysílání :

dle délky rámce a přenos. kapacity linky

(vl. zpráva endTxMsg, kind = ENDTRANSMISSION)

Další stavy:

COLLISSION: pro likvidaci „jamů“ při příjmu

konec : příjem posledního „jamu“

JAMMING: pro vysílání „ jamu “

konec : vl. zpráva endJammingMsg, kind = ENDJAMMING

BACKOFF: odročení opakovaného vysílání po kolizi

konec - vl. zpráva endBackoffMsg, kind = ENDBACKOFF

IFG: ukončení mezery mezi rámci

konec - vl. zpráva endIFGMsg, kind = ENDIFG

Příklad: CSMA / CD (viz INET)

Hrubý nástin koncepce handleMessage() v systému INET:

```
void EtherMAC::handleMessage (cMessage *msg)
{  if ( ! msg->isSelfMessage())
        { if (msg->arrivalGate() == gate("upperLayerIn"))
                    processFrameFromUpperLayer  // od vyšší vrstvy
                            (  check_and_cast<EtherFrame *>(msg));
             else  processMsgFromNetwork(msg);  // prijem ze sitě
        }   else  {  switch (msg->kind())
                        {  case ENDIFG:    // konec mezery
                           handleEndIFGPeriod();   break;

                           case ENDTRANSMISSION: konec vysílání
                           handleEndTxPeriod();   break;

                           case ENDRECEPTION: konec příjmu
                           handleEndRxPeriod();  break;

                           case ENDBACKOFF: vyčerpán  int. backoff
                           handleEndBackoffPeriod();   break;

                           case ENDJAMMING:  konec odeslání „jamu“
                           handleEndJammingPeriod();   break;
                        }
                  }
}
```

Alternativní model CSMA / CD

Model je formulován jako automat a z důvodu jednoduchosti a názornosti neobsahuje celou řadu podrobností; důraz je kladen pouze na přechodovou funkci automatu

- jde o model vrstvy přístupu ke sdílené sběrnici , který předpoládá existenci následujících portů:

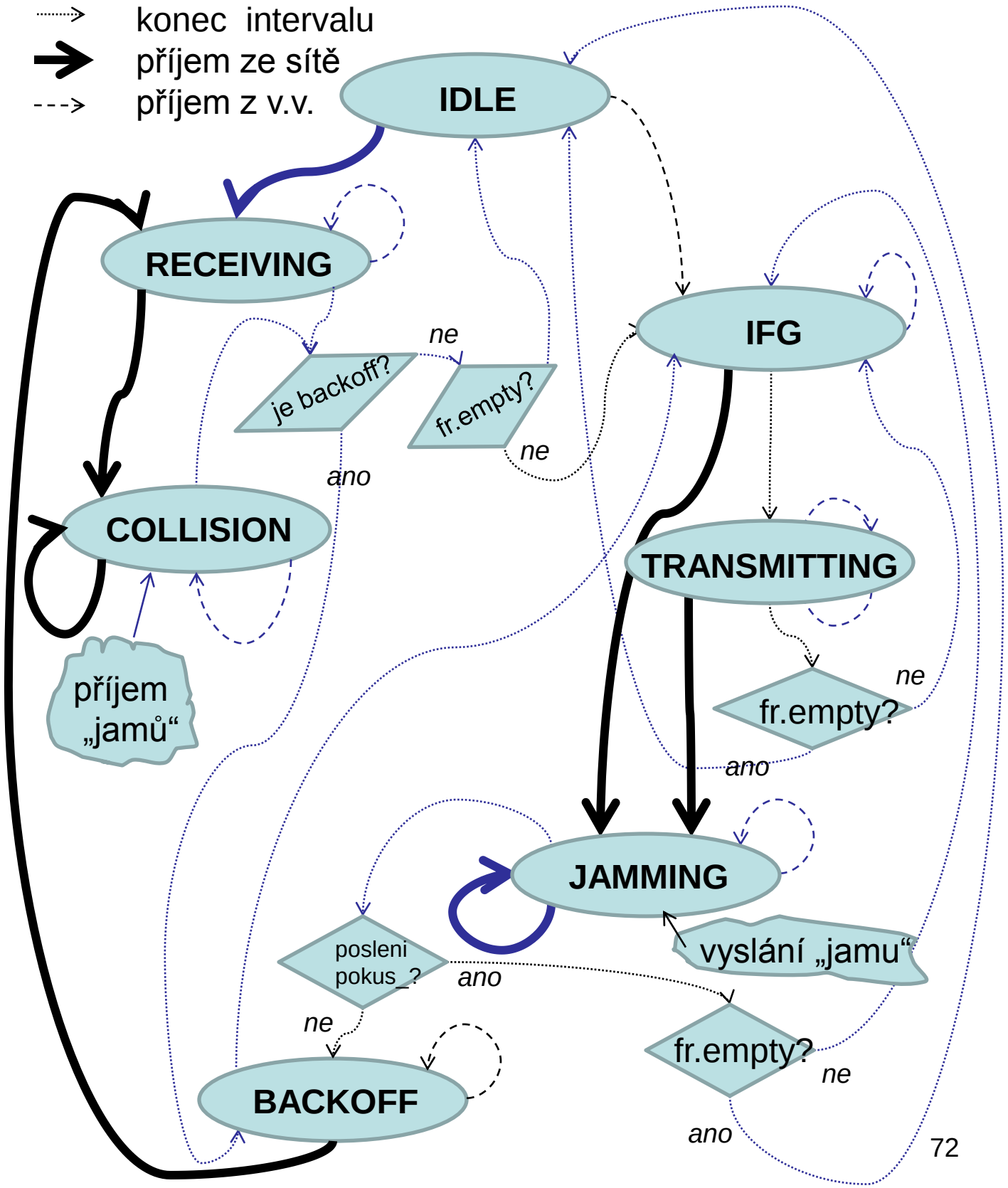
  in : upperLayerIn ; // pro příjem paketů od vyšší vrstvy,

  out: upperLayerOut; // pro předávání přijatých rámců // vyšší vrstvě

  in: physicalIn ; // pro příjem rámců ze sítě,

  out: physicalOut ; // pro vysílání rámců do sítě
- v modelu nejsou zahrnuty žádné kontroly nesprávných situací,
- není zahrnuta počáteční konfigurace stanice,
- není uvažován duplexní mód,
- přijaté pakety nejsou odesílány vyšší vrstvě,
- v modelu nejsou zahrnuty výpočty pro doby trvání různých intervalů ; jde o intervaly, jejichž konce jsou signalizovány pomocí vlastních zpráv (doba trvání intervalu backoff, doba mezery mezi dvěma vysílanými rámci, doba trvání signálu „jam“) nebo je lze odvodit z parametrů přenosové linky a přijímaného rámce (doba trvání příjmu a doba vysílání),
- model neobsahuje shromažďování dat pro statistická vyhodnocení.
- s pakety není nikterak manipulováno .

Automatový model metody CSMA/CD



Automatový popis CSMA / CD

```
class EtherMAC: public cSimpleModule
{  private: cQueue fr;        // fronta paketu od vyssi vrstvy
            cFSM fsm;        // automat
            int pocetjamu,   //  čítač očekávaných jamů
            pocetpokusu;    // počet opakovaného vysílání po kolizi

  enum  {  IDLE = FSM_Steady(0),
            RECEIVING = FSM_Steady(1),
            COLLISION = FSM_Steady(2),
            IFG = FSM_Steady(3),
            TRANSMITTING = FSM_Steady(4),
            JAMMING = FSM_Steady(5),
            BACKOFF = FSM_Steady(6),
         };
   cMessage * endRxMsg;     //  konec příjmu ze sítě
   cMessage * endTxMsg;      // konec vysílání do sítě
   cMessage * endIFGMsg;    // konec mezirámcové mezery
   cMessage * endCOLMsg;   // příjem posledního jamu
   cMessage * endBackoffMsg;   // konec  prodlevy po kolizi
   cMessage * endJammingMsg;  //  konec  vysílání jamu
   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
   Module_Class_Members ( EtherMAC, cSimpleModule, 0 );
};
```

Automatový popis CSMA / CD

```
void EtherMAC::initialize()
{  endRxMsg      = new cMessage ("konec cteni");
   endTxMsg      = new cMessage ("konec vysilani");
   endIFGMsg     = new cMessage ("konec mezery");
   endBackoffMsg = new cMessage ("konec backoff");
   endJammingMsg = new cMessage ("konec jamu");
   FSM_Goto ( fsm,IDLE );
   pocetjamu = 0;          //  čítač prijatych jamů
   pocetpokusu = 0;        //   čítač pokusů po kolizi
}

void EtherMAC::handleMessage (cMessage *msg)
{   FSM_Switch (fsm)
  {
    case FSM_Exit (IDLE):
         if (msg->arrivalGate() == gate ("upperlayerIn"))
                 {  ev << " prijem od vyssi vrstvy" << endl;
                    fr.insert (msg);
                     FSM_Goto (fsm,IFG);
                     break;
                  }
          else  {  // příjem ze sítě
                     FSM_Goto  (fsm,RECEIVING); break;
                  }
```

Automatový popis CSMA / CD

```
case FSM_Enter (RECEIVING):          //  prijem ze site
      if  ( ! endRxMsg -> isScheduled() )
                    scheduleAt( simTime() + ….., endRxMsg);
       break;
case FSM_Exit (RECEIVING):
   if (msg->arrivalGate() == gate("upperLayerIn"))   // od v.v
              { fr.insert (msg);          //  ceka na vysilani
                break;  }
   if (msg == endRxMsg)
              { ev << "cely ramec prijat v case:  "<<  endl;
                if (endBackoffMsg -> isScheduled())  // backoff
                     { FSM_Goto (fsm, BACKOFF); break; }
                if (fr.empty ()) { FSM_Goto(fsm,IDLE); break; }
                           else { FSM_Goto(fsm,IFG); break; }
              }
   if (msg == endBackoffMsg) break;   // vycerpan backoff
   if (msg->arrivalGate() == gate ("physicalIn"))   // ze site
      {  if (msg -> kind() == 1)     //  prijem „jamu“
                pocetjamu --;
         else   { delete msg;  // likvidace  přijímaného ramce
                  pocetjamu ++;  // očekávaný počet jamů
                }
         cancelEvent (endRxMsg);  // příjem přerušen
         FSM_Goto (fsm,COLLISION);  break;
      }
```

Automatový popis CSMA / CD

```
case FSM_Enter (COLLISION):  break;   //  pro likvidace jamů

case FSM_Exit (COLLISION):
    if (msg->arrivalGate() == gate("upperLayerIn"))  // od v. v.
          {  fr.insert (msg);          //ceka na vysilani
             break;   }
   if (msg == endCOLMsg)
        { if  (endBackoffMsg -> isScheduled())  // trva backoff
                    {  FSM_Goto(fsm,BACKOFF);  break;  }
          if (fr.empty ())   {  FSM_Goto (fsm,IDLE);  break; }
                    else { FSM_Goto (fsm,IFG) ;    break; }
         }
   if (msg->arrivalGate() == gate("physicalIn")) // ze site
                  {  if (msg -> kind() == 1)   // prijem jamu”
                        {  pocetjamu --;
                           if ( pocetjamu == 0) // posledni jam
                              // naplánuj konec jeho příjmu
                               scheduleAt ( simTime()+1,
                                            endCOLMsg );
                         }
                     else   pocetjamu ++;  // počet oček. jamů
                     delete msg;  // likvidace přijatého rámce
                     break;
                  }
```

Automatový popis CSMA / CD

```
case FSM_Enter(IFG):
      if ( ! endIFGMsg -> isScheduled() )
                        scheduleAt( simTime() + …, endIFGMsg);
      break;
case FSM_Exit (IFG):
     if (msg->arrivalGate() == gate("upperLayerIn"))  // od v. v.
            {   fr.insert (msg);   break; }     //ceka na vysilani
     if (msg == endIFGMsg)      // konec mezery
          {  // zacatek fyzickeho vysilánií
             send ( (cMessage *)   fr.tail() ->dup(), "physicalOut");
             pocetpokusu ++;
             break;
             FSM_Goto(fsm,TRANSMITTING);  break;    }
     if (msg->arrivalGate() == gate ("physicalIn"))
             {  if ( msg -> kind() == 1 ) pocetjamu --;      // jam
                                    else  pocetjamu ++;    // rámec
                cancelEvent  (endIFGMsg);
                delete msg;  // likvidace přijatého  rámce
                cMessage* jam = new cMessage ("jam“);
                 jam -> setKind (1);
                send (jam, "physicalOut");   // odeslání jamu na bus
                FSM_Goto (fsm,JAMMING);
                break;
             }
```

Automatový popis CSMA / CD

```
case FSM_Enter (TRANSMITTING):
        if ( ! endTxMsg -> isScheduled() )
                scheduleAt ( simTime() + ……, endTxMsg );  break;
case FSM_Exit (TRANSMITTING):
        if (msg->arrivalGate() == gate("upperLayerIn"))  // od v. v.
                { fr.insert (msg);       //  ceka na vysilani
                  break;
                }
        if (msg == endTxMsg)   // odeslan cely ramec
                { fr.pop();  // zruseni originalu po odeslani kopie
                  if (fr.empty ())  {  FSM_Goto(fsm, IDLE);  break;  }
                                else {  FSM_Goto(fsm, IFG);   break;   }
                 }
        if (msg->arrivalGate() == gate("physicalIn")) //prijem od  v.v.
                {  if ( msg -> kind() == 1 ) pocetjamu --;     // jam
                                             else  pocetjamu ++;   // rámec
                   cancelEvent (endTxMsg);  // přeruš vysílání
                   delete msg;     // zahoď  právě přijatý  rámec
                   cMessage* jam = new cMessage ("jam“ );
                   jam -> setKind (1);
                   send (jam, "physicalOut"); // výstup jamu na bus
                   FSM_Goto (fsm,JAMMING);
                   break;
                 }
```

Automatový popis CSMA / CD

```
case FSM_Enter (JAMMING):    break;
case FSM_Exit (JAMMING):
    if (msg->arrivalGate() == gate("upperLayerIn"))  //  od v. v.
              {  fr.insert (msg);            // ceka na vysilani
                 break;   }
    if (msg == endJammingMsg)     // konec vysilani jamu
          {   if (pocetpokusu <= 16)
                      {   pocetpokusu ++;
                          FSM_Goto (fsm,BACKOFF);
                      }
               else   { delete  fr. pop();  // zahozeni paketu
                        if (fr.empty ())  FSM_Goto (fsm, IDLE)     ;
                                    else   FSM_Goto (fsm, IFG);
                        break;
                      }
    if (msg->arrivalGate() == gate("physicalIn"))
            {  if ( msg -> kind() == 1 ) pocetjamu --;     // jam
                                       else  pocetjamu ++;   // rámec
              if ( pocetjamu == 0  // planuje se konec posl. jamu
                      {   scheduleAt ( simTime() + ….,
                                                 endJammingMsg );  }
              delete msg;
              break;
           }
```

Automatový popis CSMA / CD

```
case FSM_Enter (BACKOFF):;
        if ( ! endBackoffMsg  ->  isSheduled()  )
                scheduleAt ( simTime()+ … ,endBackoffMsg );
        break;

 case FSM_Exit (BACKOFF):
    if (msg->arrivalGate() == gate("upperLayerIn"))   //  od v. v.
                    {
                      fr.insert (msg);        //ceka na vysilani
                      break;
                    }
    if (msg == endBackoffMsg)
                    {
                       FSM_Goto(fsm, IFG); break;
                    }

    if (msg->arrivalGate() == gate("physicalIn"))
                    {
                      FSM_Goto (fsm,RECEIVING);   break;
                    }

  }   // konec FSM_Switch
}     // konec handleMessage
```